NAIST-IS-DT0161038

Doctor’s Thesis

Efficient Indexing Techniques for XML Data

DAO DINH KHA

February 6, 2004

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

Doctor's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
DOCTOR of ENGINEERING

DAO DINH KHA

Efficient Indexing Techniques for XML Data[*]

DAO DINH KHA

Abstract

In today's just-in-time information paradigm, the ability to communicate efficiently is vital. In this context, Extensible Markup Language (XML) has been invented as a standard for data exchange among a variety of data sources and applications. Since its invention, the popularity of the XML has been dramatically increased thanks to the ability of XML to provide a simple but standardized, extensible means to express semantic information within documents. This property makes XML possible to address the shortcomings of existing markup languages and become a key technology that facilitates information exchange for science, technology, and industries, as well as many aspects of society. XML also has been selected as the data exchange standard in newly arising business domains, such as e-business.

The semantics of XML data is richer than the one of relational data by including both of its content and metadata, the information that describes the structure of the data. Therefore, XML requires new techniques that are different from the relational databases technologies for data management. The aim of this thesis is to design efficient indexing techniques for XML data. The structural characteristics of XML data raises several challenges to design of indexing structures. In this thesis, we investigate three critical issues of XML data management as follows.

The first issue is *to cope with intensively content-updated XML data*. Among several methods of storing XML documents, a straightforward yet efficient method is to store a string representation of the XML document. An XML node is usually represented by a region coordinate, which is a pair of integers expressing the start and end positions of the substring corresponding to the node. This approach, however, has the drawback that a change of a node's region coordinate causes change of the

[*]Doctor's Thesis, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT0161038, February 6, 2004.

region coordinates of many other elements that normally degrades the performance of XML applications. To deal with the issue, we propose the Relative Region Coordinate (RRC) technique to effectively reduce the cost of recomputation by expressing the coordinate of an XML element in the region of its parent element. We present a method to integrate the RRC information into XML systems and provide experimental results.

The second issue is *to cope with structure-update of XML data*. XML queries involve both content search and structure search. For structure search, the structural information of XML data is essential to determine the structural relationships in XML documents. Several numbering schemes have been proposed so far to express the structural information using the identifiers of XML nodes. However, since the structure of XML documents can be changed, the robustness of these numbering schemes is vital for the whole indexing structure. For this purpose, we introduce a new numbering scheme called *recursive UID* (*r*UID) that has been designed to be robust in structural update and applicable to arbitrarily large XML documents. We investigate the applications of *r*UID to XML query processing in a system called SKEYRUS, which enables the integrated structure-keyword searches on XML data.

The third issue considered in this thesis is *to exploit the schematic information to improve XML query processing efficiency*. Although most of XML documents have associated DTD or XML schema, the prior query processing techniques have not utilized the document structure information efficiently. We propose a novel XML query processing method that uses DTD or XML schema to improve the I/O complexity of XML query processing. We design a Structure-based Coding for XML data (SCX) that incorporates both structure and tag name information extracted from the document structure descriptions. This property of SCX provides a Virtual Join mechanism that greatly reduces I/O workload for processing XML queries. Our experimental results indicate that SCX accelerates the processing of XML queries significantly.

XML is a new technology and its invention follows an industrial concept, where societies and industries are creating artifacts for researchers to study. Therefore, the XML-related techniques are subjects of a long development and improvement. The study presented in this thesis is an effort toward efficient XML data management.

Keywords:

Web database, XML, indexing technique, storage, query processing, numbering scheme.

Acknowledgements

This doctoral thesis describes the research results that I have achieved during my study at Database Laboratory, NAIST. I am glad to express my appreciation here to everybody who has helped me to complete the work.

I am especially indebted to Professor Shunsuke Uemura, who kindly helped me to get the scholarship of Japanese government for my graduate study in Japan and accepted me to his laboratory. During my stay at NAIST, he provides me with a great support and advices. The freedom he gave me when he allowed me to choose the subject of my research by myself made it possible for me to explore the areas which I found most appealing. I highly appreciate his kindness and cordial help.

I particularly would like to thank Professor Masatoshi Yoshikawa for introducing me to the database research topics of XML. He has provided me very useful suggestions about my research and is a critical reader of my manuscripts. I have benefited greatly from the technical discussions with him, his academic point of view and research experience. Both before and after leaving NAIST to be with Nagoya University, he has provided me with a valuable support to complete the thesis.

I also thank Doctors Toshiyuki Amasaga and Kenji Hatano for their technical help and discussions. Thanks to all members and friends of the Database laboratory for a friendly but working atmosphere. I wish them every success in their study and career.

It is a honor for me to study in such innovative university like NAIST. I want to thank the faculty and administrative staff of NAIST and Graduate School of Information Science for the excellent academic environment and research facilities. Especially, I would like to thank other members of the Thesis Committee, Professor Yuji Matsumoto and Professor Hiroyuki Seki, who spent their valuable time to referee this thesis. In the final version of this thesis, most of comments that I have received from the thesis committee have been addressed. Some of these suggestions that require a comprehensive investigation will be addressed in my future research.

Finally, I want to thank my parents and family, who always support and expect me to achieve a doctoral degree. Especially, I want to thank my wife, Ngoc Doan, and my little daughter, Ngoc Trang, who make me a happy husband and father. With their love, I managed to overcome the difficulties to complete the work. Thank you!

Dao Dinh Kha, February 2004

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Information Exchange in the Internet Era

An exciting domain of computer science is *information interchange* that is the collaboration and sharing of information among individuals, systems and applications. Unlike in the most of the 20th Century, when the task was performed mostly manually, the widespread adoption of computers in the last several decades have precipitated a paradigm shift of information interchange.

The rise of the Internet in the last decade was a big stimulus to research in this very important area by providing an excellent means for information sharing. An important fraction of the worlds computers were connected through Internet, not only by a physical network but also by a common protocol used for exchanging information. A huge amount of data has been located to be accessible by Internet and the data volume is increasing every year. With the introduction of World Wide Web, which is an application of hypertext enabling the links to documents, images, etc. anywhere on the global network, the Internet has been described as a *virtual library*. Through Internet, one can access to libraries of various disciplines, such as economy, science and technology, education, etc., to get the information used for their own purpose.

However, the massive amount of data in Internet is generated by the application of various types in different data formats. The heterogeneity poses a challenge how to interchange and integrate information from various Web sources resided in a large number of computers and information processing devices over the world. Making information truly ubiquitous over the Internet requires a common information organizing principle. Due to the variety of the Web resources, this principle had to be application-independent and easily extensible to new and unanticipated kinds of information.

For this purpose, Extensible Markup Language (XML) [47] has been designed and adopted by the World Wide Web Consortium (W3C)[1] that is the organization established to control the standardization of many WWW related formats and protocols. XML is an effort to combine the power of Standard General Markup Language (SGML)[43] and the simplicity of implementation of HyperText Markup Language (HTML)[44]. XML is a subset of SGML which is formulated by simply removing the features which are rarely used or cause problems in terms of processing speed. Although some simplifications have been made, XML still provides an enriched data expressiveness for presenting various kinds of information.

XML is perceived as a standard with the potential of replacing HTML, which is the format for rendering most of current web pages. Today's search engines are mostly based on keywords appearing in the content pages rather than on the meaning of the search query. A search on specific words returns documents that may include the information that is not of interest. To solve this problem, XML allows user-defined tags to add semantics to the Web content page. In addition, XML allows the separation of data and presentation in a document. As a result, the same data can be presented in different formats according to particular demands.

XML enables computer systems to work together through the exchange of information items, from a number to complex data structures. To be cross-platform compatible, XML provides a *neutral* notation for labeling the parts of information and representing the relationships among these parts. Therefore, the applications that agree on a common XML vocabulary can perform data interchange. For instance, Web browsers can be compatible with drawing tools like Corel Draw and Acrobat Illustrator using an XML-based language called the Scalable Vector Graphics Language[45]. Similarly, XML is also a metalanguage, i.e. a set of tags and the rules connecting the tags and their contents, for defining other markup languages such as Real Estate Listing Markup Language[2], Wireless Markup Language[3], or Open Financial Exchange[4].

Furthermore, XML is used as the core for development platforms of an entire appli-

[1]http://www.w3.org/
[2]http://xml.coverpages.org/openMLS.html
[3]http://www.oasis-open.org/cover/wap-wml.html
[4]http://xml.coverpages.org/ofx.html

cation such as Microsoft *.NET*, where XML serves as a means to model data-integrated components of information systems. Since XML does not mandate any particular storage technique, it enables the information interchange among systems that store data in various formats, such as file systems, relational databases, object repositories, etc.

1.2 Markup Languages and XML's Originality

XML comes from a rich history of markup languages used for text processing. Text processing is a sub-discipline of computer science dedicated to creating computer systems that can automate parts of the document creation and publishing process. A markup language specifies what markup is allowed, how markup is distinguished from data, and what markup means. Markup text added to the data content of a document in order to convey information about data to computers. There are several requirements imposed on markup languages. First, various computer programs and systems should be able to read and write markup data. Second, the markup language must be extensible to support different types of information. Third, there must be a mechanism for formally describing the rules shared by documents of a common type or class.

Normally, a markup language aims at a specific application, such as document publishing, document's information interchanging, or document rendering. There are a number of markup languages as follows:

Portable Document Format *(PDF)*. This is a format proposed by Adobe Systems, Inc.[5] for publishing works. PDF enables a consistent look of published documents across different platforms. PDF is the most preferable format for storing and submitting a vast number of documents. However, PDF does not preserve structure and contextual information about a document. Since PDF does not separate rendering and content, the language is unsuitable for many data-driven application scenarios. In addition, PDF requires a proprietary reader, hence indexing and querying are ruled out.

Standard Generalized Markup Language. SGML is a text-based data format adopted as an international standard (ISO 8879) for the markup of electronic structured documents. SGML has been accepted in the professional publishing industry and used successfully in many large-scale systems. SGML enables the creation of documents with

[5]http://www.adobe.com/

varying degrees of structure and contextual information. The problem with SGML is that it is *complex*. SGML based Internet publishing requires that all participating parties use SGML systems in their existing infrastructure. The complexity of SGML and the effort it takes to build SGML systems makes the use of SGML very costly.

Hypertext Markup Language. HTML is an application of SGML that was specifically designed for rendering data in documents to be published on the World Wide Web. The success of HTML can be attributed to its restricted and easy understandable vocabulary that makes it supported by numerous tools. However, HTML rendering information is limited and there is very little opportunity to include information about the document and its hierarchy. A document in SGML format can be translated to HTML for publishing on the Internet with the cost of its semantic richness.

XML's originality. Due to its complexity, SGML is not widely used and was broken down on the acceptance criteria to design XML. The work was initially developed by an XML Working Group formed in 1996 under W3C, which received input from an XML Special Interest Group. XML is an essential core of SGML and optimized for the processing and information exchanging by Internet by removing some features of SGML. For instance, the content model minimization parameters, markup delimiters and name characters definition, char-set declaration, and optimal features turning of SGML are all eliminated in XML to reduce the parsing workload. Whereas SGML allows many situations where it is difficult for the parser to understand the separation between markup and other types of characters, XML's errors of the parsing a document is more understandable. The declarations that modify the allowed subtree of an element type in SGML are eliminated in XML to make the document type declaration simpler. The AND (&) content model groups, that generates factorial of the number of elements combinations of OR operations in SGML is eliminated in XML, etc. The changes enable XML data to be processed more efficiently. The simplification in XML specification is probably one of the main reasons behind the wide support of XML.

1.3 The Significance and Research's Objectives

Since the semantics of XML data is represented by both of its content and metadata, there are differences between XML data and relational data in terms of data model and usages, as briefly described bellow:

- *Flat relational data and nested XML data*, and the depth of nesting can be irregular. Nested data structures can be represented by using tables with foreign keys in relational databases, but it requires a costly recursive operation to search these structures for objects at an unknown depth of nesting. In XML, such a search is very common. For example, the query "*Find all materials published or produced in 2003*" on the XML data describing the library properties of a company can be represented by the expression `//*[@year = "2003"]`. This query in a relational query language requires a more complex expression due to the lack of the type of material and the explicit hierarchical information.

- *Homogeneous relational data and heterogeneous XML data*. Relational data is organized in the form of a table of tuples, each of them has the same attributes, with the same names and types. This allows metadata to be defined using fixed schemes and removed from the data itself to a separate catalog. In contrast, XML's metadata is distributed throughout the data in the form of tags. Each instance of an XML element can have a different structure that can be changed. Therefore, XML queries can naturally involve both data and metadata, where in a relational language such queries would require a join of several data tables and system catalogs.

- *Data-derived order and document order*. In a relational database, the order of the rows of a table, if exists, is determined using their values. XML documents, on the other hand, have an intrinsic document order that can be important to their meaning and is independent from data values.

In data management, a query language is a set of rules that specifies the way data is selected and reorganized on demands whereas indexing structures are the way data is organized physically in order to perform queries efficiently. The role of indexing structure is increasing when the amount of data to be queried is large. The traditional indexing structures such as B$^+$-tree and R-tree have been designed to deal with the data represented by the relational data model. As briefly presented in Appendixes A and B, the order of data in these structures is not considered and derived artificially based on the value of data items. From the above-mentioned differences between relational data and XML data, it is natural that we need to extend the traditional techniques and develop new ones specific for XML data management.

Research Objectives. This thesis aims at proposing the indexing structures that are efficient for XML query processing. A big challenge to an XML indexing structure is that it must encompass the semantics feature of XML data, i.e. it must keep the original order of XML elements that may be possibly established at several hierarchical levels. However, the order may not be derived from the data value. Therefore, an XML indexing structure must keep information of both data value and document hierarchy of XML documents. The efficient integration of these features is not trivial task if we take into account all problems that may emerge. In this thesis, we investigate three specific issues for XML data indexing, namely:

Coping with intensively content-updated XML data

XML data items have various lengths tailored in a tree structure, hence content updates to XML data may change the position of data items in their original XML documents and the change may affect the structure of indexes on the updated data. We investigate how to reduce the cost of the content updates.

Making indexing structure robust on structure-update of XML data

Since indexing structures are based on the identifiers of XML elements but the structure of XML documents can be changed by node insertion or deletion. We propose an indexing structure that is robust on structural updates of any type.

Using schematic information for improvement of XML query processing

Schematic information, or document structure information, normally is expressed in document type definition associated with most of XML documents. We investigate how to exploit the information to establish the hierarchical order among the XML elements efficiently to improve XML query processing.

1.4 Thesis Outline and Contribution

The thesis has six chapters and appendixes. After this introductory chapter, in Chapter 2 we review the specification of XML in order to make the thesis self-contained. The XML-related languages that have been referred in our research are also reviewed here.

Chapter 2 is concluded by the description of the data model and the common notation used in the thesis.

The main result of our research that answer the above mentioned objective issues is presented in Chapters 3, 4, and 5. In Chapter 3, we describe an approach using relative region coordinate for the applications that have to deal with heavily content updated XML data. In Chapter 4, we present an indexing structure based on a recursive numbering scheme that is robust on structural update. In Chapter 5, we describe an indexing structure that significantly improves the I/O workload of XML query processing. In order to make the document easy to be read, in each chapter we review the work related to the issue addressed in the chapter.

Finally, Chapter 6 concludes the main content of the thesis by a summary of the main results and a discussion about the future development of our current research. Some auxiliary information related to the research can be found in the Appendix.

Chapter 2

Preliminaries

This chapter introduces the essential part of the XML specification in order to make the thesis self-contained. Besides the most important notions of XML, we but also describe the relation of the material reviewed here and the research issues addressed in the following chapters. The details of the XML concept is covered in depth in the home pages of W3C. The first section of the chapter describes the basic of XML specification. The second section reviews XML-related standards that are directly or indirectly used in our research. The third section gives an overview of issues relating to XML data management. The last section presents the notation commonly used in the next chapters of the thesis.

2.1 Extensible Markup Language

The invention of XML follows the industrial concept, where societies and industries are creating artifacts as standards based on demands. The main features of XML are the usability over the Internet, the ability to support a wide variety of applications, the compatibility with SGML, and the simplicity for processing and creating.

XML has been designed taking into account the existing data formating standards such as SGML and HTML. Consider, for instance, the simple HTML document in Listing HE.1 bellow. The data contained in the document describes some information about a person and how the information will appear to the viewers through the formatting tags, namely HTML tags, in a web page. There is no semantic information that the data represents a person's name and e-mail address.

Listing HE.1: a HTML document

```
<html>
<head>
    <title>Personal Information</title>
</head>
<body>
<p> <b>Name: </b>Branson Richard</p>
<p> <b>Email: </b>branson@domain.com</p>
</body>
</html>
```

XML can address the above problem, as the data in an XML document is self-describing. XML allows defining custom tags describing the data enclosed by them. An example XML document containing data about the same person is shown in Listing XE.1.

Listing XE.1: an XML document

```
<?xml version="1.0"?>
<!DOCTYPE person SYSTEM ‘‘person.dtd’’>
<person status = ‘‘manager’’>
<name>
    <family>Branson</family>
    <given>Richard</given>
</name>
<email>branson@domain.com</email>
</person>
```

Unlike in HE.1, in XE.1 the data’s semantics can be interpreted from the document through the tags `<person>`, `<name>`, etc, where the names of these tags are chosen in the understanding of the document’s creator to interpret the document’s content.

2.1.1 XML document specification

The specification of XML is descriptively identified in the *XML 1.0 W3C Recommendation*. An XML document is composed of *declaration*s, *element*s, *comment*s, *character reference*s, and *processing instruction*s, all of which are indicated in the document by explicit *markup*. The function of the markup in an XML document is to describe its storage layout and logical structure and to associate attribute-value pairs with its logical structures. The markups are defined flexibly, that makes XML an ideal basis for defining arbitrary languages. For example, the XML schema language, which will be presented in Section 2.1.2, used to describe the structure of XML documents is based on XML itself.

A software module called an *XML processor* is used to read XML documents and provide access to their content and structure. An XML processor is doing its work on behalf of another module, called the application, according to the XML specification.

The Logical and Physical Structures of XML

Each XML document has both a logical and a physical structures that are presented by the concepts of *element* and *entity*, respectively.

Logical structure. Each XML document contains one or more elements, the boundaries of which are either delimited by *start-tag*s and *end-tag*s, or, for empty elements, by an *empty-element tag*. Each element has a type that is identified by name. In the XML document in Listing XE.1, `<family>Branson</family>` is an element and `<family>` and `</family>` are start-tag and end-tag of the element, respectively.

An element may have a set of *attribute*s that occur inside element start tag, after element type name. Element can contain only one occurrence of each attribute. Each attribute specification has a name and a value. For example, the attribute **`status`** is specified in the element **`person`** like `<person status = ‘‘manage’’>`.

All text that is not markup constitutes the character data of the document. The text between the start-tag and end-tag is called the element’s content. The notion of content of an element will be explained in details in Section 3.2 to describe the issue of content update of XML data. An element with no content is said to be empty. The representation of an empty element is either a start-tag immediately followed by an end-tag, or an empty-element tag.

XML documents should begin with an *XML declaration*, which must appear before the first element in the document. The document type declaration specifies the version of XML being used. For example, the XML document in Listing XE.1 starts with the declaration `<?xml version=‘‘1.0’’?>`.

Well-formedness. XML syntax is constrained by a grammar that governs the permitted tag names, attachment of attributes to tags, and so on. All XML documents must conform to these basic grammar rules. A textual object is a *well-formed* XML document if it contains one or more elements and there is exactly one element, called the *root*, or document element, no part of which appears in the content of any other element. For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other. As a consequence of this, each non-root element has the unique parent element. Such conformed documents can be interpreted by a common XML processor.

Physical structure. An XML document is defined as a series of characters that can be organized *non-linear* and potentially in multiple pieces of text. The *piece-of-text* construct is called an *entity*. An XML document may consist of one or many storage units called entities. Whereas XML elements describe the logical structure of XML document, entities keep track of location of chunk of bytes that make up an XML document. Therefore, entities are called *physical structure* of the document.

An entity may refer to other entities to cause their inclusion in the document. A document begins in a *root* or document entity, which serves as the starting point for the XML processor and may contain the whole document. An entity contains either *parsed* or *unparsed* data. Parsed data is made up of characters, some of which form character data, and some of which form markup. A parsed entity's contents are referred to as its replacement text; this text is considered an integral part of the document. An unparsed entity is a resource whose contents may or may not be text, and if text, may be other than XML. Each unparsed entity has an associated notation, identified by name.

The XML parser programs, including that ones used in our experiments, must understand the physical structure of XML documents in order to establish correctly their hierarchy. The presentation of the hierarchy in main memory and to users is discussed in section 2.2.6.

2.1.2 XML Document Types and Schemas

Both Document Type Definition (DTD) and XML schema are mechanisms used to define the structure of XML documents. They determine what elements can be contained within the XML document, how they are to be used, what default values their attributes can have, and so on. Given a DTD or XML schema and its corresponding XML document, a parser can validate whether the document conforms to the desired structure and constraints. This is particularly useful in data exchange scenarios as DTDs and XML schemas provide and enforce a common vocabulary for the data to be exchanged.

Document Type Definition. A *markup declaration* is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration. For example, `<!ELEMENT person (name, email?, person*)>` is a markup declaration.

The XML document type declaration contains a list of markup declarations of elements and attributes that provide a grammar, called Document Type Definition or DTD, for a class of documents. The document type declaration also can point to an external subset (a special kind of external entity of its physical structure) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.

Listing XD.1 shows a DTD for the XML document in Listing XE.1. It describes which primitive elements form valid components for the three composite ones: **`person`**, **`name`**, and **`email`**. The keyword `#PCDATA` signifies that the element does not contain any tags or child elements and only parsed character data.

Listing XD.1: a DTD

```
<!ELEMENT person (name, email?, person*)>
<!ATTLIST person status CDATA #IMPLIED>
<!ELEMENT name (family, given)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

Suppose that the DTD is contained in the file named “person.dtd”, in the XML document XE.1, the DTD is declared by the following:

```
<!DOCTYPE person SYSTEM ‘‘person.dtd’’>
```

XML Schema. XML schemas differ from DTDs in that the XML schema definition language is based on XML itself. As a result, the set of constructs available for defining an XML document is extensible. XML schemas support more complex structures than DTDs. In addition, stronger typing constraints on the data enclosed by a tag can be described because primitive data types such as string, decimal, and integer are supported. This makes XML schemas suitable for defining data-centric documents.

Listing XS.1: an XML schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
 <xs:element name='email' type="xs:string"/>
 <xs:element name='family' type="xs:string"/>
 <xs:element name='given' type="xs:string"/>
 <xs:element name='name'>
  <xs:complexType>
   <xs:sequence>
     <xs:element ref='family'/>
     <xs:element ref='given'/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name='person'>
  <xs:complexType>
   <xs:sequence>
     <xs:element ref='name'/>
     <xs:element ref='email' minOccurs='0'/>
     <xs:element ref='person' minOccurs='0' maxOccurs='unbounded'/>
   </xs:sequence>
  <xs:attribute name='status'/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Listing XS.1 shows an XML schema for the XML document in Listing XE.1. The *sequence tag* is a compositor indicating an ordered sequence of subelements. There are other compositors for *choice* and *all*. As shown for the **`person`** element, it is possible to constrain the minimum and maximum instances of an element within a document.

XML schema definitions can exploit the same data management mechanisms as designed for XML because an XML schema is an XML document itself, whereas DTDs require specific support to be built into an XML data management system.

Validation. In addition to being well formed, the structure of a particular XML document can be validated against a Document Type Definition (DTD) or an XML schema. An XML document is *valid* if it has an associated document type declaration and if the document complies with the constraints expressed in it. XML document type definition are used to describe constraints on the logical structure and to support the use of predefined storage units.

2.2 Main XML-related standards

There is a large set of XML-related standards presented in the form of W3C recommendations. A review of all of them is out of the scope of this thesis. In this part, we review briefly the standards, the concepts of which are used in our research, such as XML Namespaces, XML Information Set, XML Path Language, Extensible Stylesheet Language and Extensive Stylesheet Language Transformation, XML Linking Language, Document Object Model and Simple API for XML.

2.2.1 XML Namespaces

There is a problem that arises when XML allows anybody to choose the tag name of XML elements and attributes, since the same name may be used with different meanings. This makes it very difficult to work with documents from independent sources. There is a standard called *XML Namespaces* to solve the problem.

A namespace is an area within which a name have the same meaning whenever it is used. Formally, an XML namespace is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names. Names from XML namespaces are *qualified* names consisting of a *namespace prefix* and a *local part* separated by a single colon, where the prefix mapped to a URI reference selects a namespace. The combination of the universally managed URI namespace and the document's own namespace produces identifiers that are universally unique.

A namespace is declared using a family of reserved attributes. Such an attribute's name must either be `xmlns` or have `xmlns:` as a prefix. Bellow is an example of namespace declaration, which associates the namespace prefix `edi` with the namespace name `http://ecommerce.org/schema`. The `edi` prefix is bound to the namespace within the `x` element. The qualified name `edi:price` serves as an element type.

```
<x xmlns:edi='http://ecommerce.org/schema'>
  <edi:price units='Euro'>32.18</edi:price>
</x>
```

2.2.2 XML Information Set

XML Information Set (Infoset) is an abstract data set, the purpose of which is to provide *a consistent set of definitions* for use in other specifications that need to refer to the information in a well-formed XML document. An XML document has an information set if it is well-formed and satisfies the namespace constraints.

The information set for any well-formed XML document contains exactly one a document information item and several other information items of the types: Document, Element, Attribute, Processing Instruction, Unexpanded Entity Reference, Character, Comment, Document Type Declaration, Unparsed Entity, Notation, Namespace. An information item is an abstract description of some part of an XML document and has a set of associated named properties. Information items are directly accessible from the properties of the document information item or through the properties of other information items. For example, the description of a document information item may have the following properties.

children A document-ordered list of child information items including exactly one element information item, processing instruction and comment information items.

document element The element information item corresponding to the document element.

notations A set of notation information items, one for each notation declared in DTD.

unparsed entities A set of unparsed entity information items, one for each unparsed entity declared in DTD.

base URI The base URI of the document entity.

character encoding scheme The name of the character encoding scheme in which the document entity is expressed.

standalone An indication of the standalone status of the document that is derived from the optional *standalone* document declaration at the beginning of the document entity.

version A string representing the XML version of the document.

2.2.3 XML Path Language

Information within an XML document can be located through a language called XML Path Language (XPath)[48], the primary purpose of which is to address parts of an XML document. The notion of XPath will be used extensively in chapters 4 and 5.

XPath can refer to textual data, elements, attributes, and other information in an XML document in two ways: A *hierarchical fashion based on the ordering of elements in a document tree*, and an *arbitrary manner relying on elements having unique identifiers*. In addition, XPath has a subset that can be used for testing whether or not a node matches a pattern. It also provides basic facilities for manipulation of strings, numbers and boolean in the logical structure of an XML document.

XPath operates on the abstract, logical structure of an XML document. XPath models an XML document as a tree of nodes of different types, including element, attribute, and text nodes. The primary syntactic construct in XPath is the *expression* that is evaluated with respect to a context to yield an object, which has one of the following four basic types: 1) node-set; 2) boolean; 3) number; and 4) string.

One important kind of XPath expression is *location path*. A location path selects a set of nodes relative to a context node. The result of evaluating a location path is the node-set containing the nodes selected by the location path. Location paths can recursively contain expressions that are used to filter sets of nodes. The core rules of XPath are follows:

```
[1]   LocationPath ::= RelativeLocationPath | AbsoluteLocationPath
[2]   AbsoluteLocationPath ::='/'RelativeLocationPath?
[3]   RelativeLocationPath ::= Step | RelativeLocationPath '/' Step
```

A location step has three parts: 1) an axis, which specifies the hierarchical relationship between the nodes considered in the location step and the context node; 2) a node test, which specifies the node type and expanded-name of the nodes selected by the location step, and 3) zero or more predicates to further refine the set of nodes. An initial node-set is generated from the axis and node test, and then filtered by each of the predicates in turn. A predicate filters a node-set with respect to an axis to produce a new node-set. The available XPath axes are: *ancestor*, *ancestor-or-self*, *attribute*, *child*, *descendant*, *descendant-or-self*, f*ollowing*, *following-sibling*, *namespace*, *parent*, *preceding*, *preceding-sibling*, and *self*. Here are some examples of location paths:

- `child::para` selects the para element children of the context node.
- `child::* selects` all element children of the context node.
- `/child::doc/child::chapter[position()=5]` selects the fifth chapter of the doc document element.
- `child::*[self::chapter or self::appendix][position()=last()]` selects the last chapter or appendix child of the context node.

Location path also can be described using abbreviated syntax. For example, **`para`** selects the 'para' element children of the context node, or **`*`** selects all element children of the context node.

Note that from XPath specification, it is clear that a typical operation of querying XML data is looking for elements or attributes having a given type, i.e. a name. This is probably one of a primary reason behind the introduction of the concept *structural join* that will be discussed in chapter 5.

2.2.4 XSL and XSL Transformations

Since XML documents do not contain any rendering information, they can be formatted in a flexible manner. A standard approach to formatting XML documents is using XSL, the eXtensible Stylesheet Language [49]. XSL specification is composed of two parts: *XSL Formatting Objects* (XSL FO) and *XSL Transformations* (XSLT). XSL FO provides formatting and flow semantics for rendering an XML document . A rendering agent is responsible for interpreting the abstract constructs provided by XSL FO in order to instantiate the representation for a particular medium. XSLT is a language that can transform XML documents into other XML documents. A transformation in the XSLT language is expressed as a well-formed XML document called a *stylesheet* describing template rules for transforming.

A template rule has a pattern which is matched against nodes in the source XML tree and a template which can be instantiated to form part of the result XML tree. A template can contain elements that specify literal result element structure. A template can also contain elements from the XSLT namespace that are instructions for creating result tree fragments. When a template is instantiated, each instruction is executed and replaced by the result tree fragment that it creates. Instructions can select and process descendant source elements. Processing a descendant element creates a result tree fragment by finding the applicable template rule and instantiating its template. The result tree is constructed by finding the template rule for the root node and instantiating its template. In constructing the result tree, elements from the source tree can be filtered and reordered, and arbitrary structure can be added.

Listing XT.1: an XML transformation

```
<?xml version=‘‘1.0’’?>
<xsl:stylesheet xmlns:xsl= ‘‘http://www.w3.org/1999/XSL/Transform’’
version=‘‘1.0’’>
<xsl:template match="/">
<html>
<head><title>Personal Information</title></head>
    <body>
    <xsl:apply-templates select=‘‘person/name’’/>
    </body>
    </html>
    </xsl:template>
    <xsl:template match="name">
    <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="family">
    <p><b><xsl:text>Family name: </xsl:text></b>
    <xsl:value-of select="."/></p><br/>
    </xsl:template>
    <xsl:template match="given">
    <p><b><xsl:text>First name: </xsl:text></b>
    <xsl:value-of select="."/></p>
    </xsl:template>
</xsl:stylesheet>
```

Although designed to transform an XML vocabulary to an XSL FO vocabulary, XSLT can be used for a range of transformations including those to HTML. The example stylesheet that uses a set of simple XSLT templates and XPath expressions to transform a part of the XML document XE.1 to HTML is shown in Listing XT.1 and the resulted HTML document is shown in Listing HE.2.

Listing HE.2: the resulted HTML document

```
<html>
    <head>
    <title>Personal Information</title>
    </head>
    <body>
    <p><b>Family name: </b>Branson</p>
    <br>
    <p><b>First name: </b>Richard</p>
    </body>
</html>
```

In XT.1, the declarations `</xsl:template>` and `</xsl:template>` indicate the start and end of the application of a template. The element `family` in Listing XE.1 is checked by the declaration `<xsl:template match=‘‘family’’>`. The code `<b><xsl:text>Family name:  </xsl:text></b>` provides a format and the code `<xsl:value-of select=‘‘.’’/>` determines the content of the output.

2.2.5 XML Linking Language

The notion of *resource*s, which are addressable units of information or service, is universal to the World Wide Web. A resource is addressed by a Uniform Resource Identifier (URI) reference. XML Linking Language (XLink)[50] is an explicit relationship between resources or portions of resources. The relationship is expressed by an XLink linking element, which is an XLink-conforming XML element that asserts the existence of a link. Even though XLink links must appear in XML documents, they are able to associate all kinds of resources including files, images, documents, programs, and query results. It is possible to address a portion of a resource, for example, a particular element inside an XML document.

A common use of XLink is to create a hyperlink, which is a link used primarily for presentation to a human user. A simple XLink link is shown in the listing bellow:

```
...for more information, consult:
<citation xlink:type "simple"
          xlink:href="http://www.uw.ca/paper.html"
Biesman (1977)
</citation>
```

2.2.6 Document Object Model and Simple API

For interoperability and modularity, an XML processor must represent XML data in common rules called API that can be understood by other program modules. There are two main API*s* for manipulating XML documents in an application. They are now part of the Java API for XML Processing (JAXP version 1.1).

Document Object Model (DOM) is a W3C's standard operating system and programming language–independent model for manipulating hierarchical documents in memory. A DOM parser parses an XML document and builds a DOM tree, which can then be used to traverse the various nodes. However, the whole tree has to be constructed before traversal can commence. As a result, memory management is an issue when manipulating large XML documents. This highly resource intensive feature is visible especially in cases where only a small section of the document is to be manipulated.

Simple API for XML (SAX) bases on an event-driven model. Each time a start or end tag, or processing instruction is encountered, the program is notified. As a result, the whole document does not need to be parsed before it is manipulated. In fact, sections of the document can be manipulated as they are parsed. Therefore, SAX is better suited to manipulating large documents as compared to DOM.

XML parser tools. Among a number of freeware programs written to promote the popularity of XM usage, the most popular are XML parsers. For example, Apache XML Project from the Apache Software Foundation[4] offers both DOM and SAX parsers called Xerces that provides highly modular and configurable XML parsing and generation. Fully-validating parsers are available for both Java and C++, implementing the W3C XML and DOM standards, as well as the SAX standard. In our experiments performed in this study, both Xerces DOM and SAX parsers are used.

2.3 XML Data Management

To give a broader picture of the XML technology's development, we review the current trends and issues of XML data management. As more and more organizations and systems employ XML for their information management and exchange activities, the issues related to XML's efficient and effective storage, retrieval, querying, indexing, and manipulation emerge. In a higher level, organizations and system developers concerning with XML data management have to answer the questions like that what are the efficient XML data management solutions? What are the features, services, and tools XML data management systems must have? Which XML data management system or approach is the best in terms of performance and efficiency for a particular application? Are there any good practice and domain or application-specific guidelines for information modeling with XML, and so on.

Database vendors have responded to these new data and information management requests. Most commercial database systems offer extensions and plug-ins to support the management of XML data. In addition to extending existing database management systems to support XML, native XML databases have been invented.

There are several approaches to implement a system to manage XML data. XML documents fall into two broad categories: *data-centric* and *document-centric*. Data-centric documents are those where XML is used as a data transport. They include sales orders, patient records, and scientific data. Their physical structure, for instance, the order of sibling elements, is often unimportant. A special case of data-centric documents is dynamic Web pages, such as online catalogs and address lists, which are constructed from known, regular sets of data. Document-centric documents are those in which XML is used for its SGML-like capabilities, such as in user's manuals, static Web pages, and marketing brochures. They are characterized by irregular structure and mixed content and their hierarchical structure is important.

To store and retrieve the data in data-centric documents, the kind of need software depends on how well structured your data is. For highly structured data, such as the white pages in a telephone book, an XML-enabled database, such as a relational or object-oriented database together with some sort of data transfer software, is suitable. This may be built in to the databases, called XML-enabled, or might be third-party software, such as middleware or an XML server. If the data is semi-structured, such as the yellow pages in a telephone book or health data, both of relational databases or native XML databases are choices.

To store and retrieve document-centric XML documents, a native XML database or content management system is suitable. Some XML-enabled databases provide native storage as well such as Oracle 9*i*[28] or DB2 Extender[23]. These are designed to store content fragments, such as procedures, chapters, and glossary entries, and may include document metadata, such as author names, revision dates, and document numbers.

Content management systems, the core of which is a native XML database, normally have additional functionality, such as editors, version control, and workflow control.

2.3.1 XML and database products

The XML products can be divided into the following categories, the boundaries between some of these categories are somewhat fuzzy.

1. *Middleware*: Software you call from your application to transfer data between XML documents and databases.

2. *XML-Enabled Databases*: Databases with extensions for transferring data between XML documents and themselves. An approach is to use object-oriented databases presenting components of an XML document as objects. Another approach is to use RDBMS with extended XML-supporting functions.

3. *Native XML Databases*: Databases that store XML in *native* form, generally as some variant of the DOM mapped to an underlying data store.

4. *XML Servers*: XML-aware J2EE servers, Web application servers, integration engines, and custom servers. For data- and document-centric applications.

5. *Wrappers*: Software that treats XML documents as a source of relational data and uses SQL to query XML data.

6. *Content Management Systems*: Applications built on top of native XML databases and/or the file system for content/document management. Include features such as check-in/check-out, versioning, and editors.

7. *XML Query Engines*: Standalone engines that can query XML documents.

8. *XML Data Binding*: Products that can bind XML documents to objects. Some of these can also store/retrieve objects from the database.

2.3.2 XQuery for XML query processing

Since XML is emerging as a standard format for data interchange among applications, there is increasing amounts of information of many different stored using the XML format. Therefore, the ability to efficiently query XML data sources across application boundaries and diverse sources becomes increasingly important. The application-independence of query processing means if an application is viewed as a source of information in XML format, queries must be formed based on that XML format only, not on the application itself.

XQuery[51] is designed to meet the requirements identified by the W3C XML Query Working Group stated in *XML Query 1.0 Requirements* and the use cases stated in *XML Query Use Cases*. XQuery provides a flexible facility to extract information from real and virtual XML information sources, including both databases and documents. XQuery operates on the logical structure of an XML document as the Query data model, which represents XML data in the form of nodes and values. XQuery is closed under the Query data model, which means that the result of any valid XQuery expression can be represented in this model. XQuery is derived from an XML query language called Quilt, which in turn borrowed features from several other languages, including XPath 1.0, XQL, XML-QL, SQL, and OQL. Currently, XQuery is a working draft submitted to W3C to be approved.

2.4 Data Model and Notation used in this thesis

This work's notation and terminology are supposed to be standard. However, here are a few conventions that hold throughout the following chapters.

Data Model. In this thesis, we present an XML document as a labeled rooted tree, called the *XML tree* of the document. The tree is abstracted out from the DOM data model [46] and similar to the data model used in XPath specification. A node may be one of different node types: *root*, *element*, *attribute*, *text*, etc. [47]. A node is the parent of another node if the element corresponding to the former node contains the element, or attribute, or text corresponding to the latter node.

Common Notation. In this thesis, hereafter, we use the following common notation to describe the investigated issues:

- **T** denotes an XML tree rooted at the root node **r**,
- **m**, **n**, **p**, **q**, etc, are nodes of **T**,

- $\mathscr{P} \equiv (\mathbf{r} = \mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2, ..., \mathbf{n}_k = \mathbf{n})$ is the simple node path for **n** from **r**. The *level* of a node is the length of the node path for the node. For example, the level of **n** is the length of $\mathscr{P}$, i.e., k.

- the parent node of a node **n** is determined by *parent*(**n**),

- the tag name of **n** is determined by **n**.*tag*

Since XQuery is a work in progress to be approved by W3C, its design is a subject to change. On the other hand, XPath has been approved and published as a W3C recommendation. Therefore, in this thesis our discussion on indexing techniques will be based on the XPath notion.

Finally, to make the material easy to read, XML documents used in the examples presented in this thesis conform the same DTD. However, for simplicity, a part or the entire of the DTD may be used depending on the context. Specifically, the DTD used in the examples in Chapter 3 is a part of the DTD used in Chapter 5.

Chapter 3

XML Content Update using Relative Region Coordinates

3.1 Introduction

An important feature of the XML is the ability to manage data items having various lengths tailored in a tree structure. XML documents can be viewed in two ways: a tree and a character string. Several methods to store and index XML data have been proposed so far [33][24][12][26][6][14][5]. A simple yet efficient approach is to store a string representation of the XML document, e.g. [26][6][14]. A node in an XML tree corresponds to a substring of the entire string of the XML document. Such substrings can be naturally identified by a *region coordinate*, which is a pair of integers representing the start and end positions of the substring counting from the start of the XML document.

An important role of region coordinates is fast extraction of substrings representing an XML subtree. Because substrings of answers for a query can be retrieved directly from the original documents, the use of region coordinates supports low-cost publishing of XML data. Typical XML queries search tree nodes satisfying a given structural condition and returns substrings for the XML documents. The approach of storing string representation of XML documents and region coordinates is suitable for processing such typical XML queries.

However, a major drawback of this approach is that it is not robust against content updating. Since absolute region coordinates are used to represent node positions, a simple update, such as an insertion (or deletion) of a word, causes a change in the region coordinate values of the successive nodes in the document. If the insertion (or deletion) is made near the start of the document, the region coordinates of almost all nodes need to be updated. If the update frequency is high and the size of XML data is large, the recomputation of the region coordinates of nodes in the index data degrades system performance. The robustness of region coordinates in XML updating has not been adequately addressed in the previous works.

3.1.1 Contribution

In this chapter, we present a technique, called *Relative Region Coordinate* (*RRC*), that efficiently deals with the content update problem. The main idea of this technique is to express the region coordinate of XML elements based on their parent or the appropriately selected ancestor elements. Using RRC, the workload needed to recompute the region coordinates in the indexing data when content updates occur is greatly reduced. We also show that query processing time with RRC is comparable to that with normal region coordinates.

In principle, the idea of RRC is applicable to any method employing region coordinates. Therefore, there are several approaches to implementing the RRC technique in XML databases. In this chapter, we demonstrate the applicability of RRC to relational database systems storing XML documents. A native XML indexing structure based on the RRC technique can be found in [9]. This research's result can be applied in the applications categorized into the *XML-Enabled Databases* group, as discussed in Section 2.3.1, page 23 of this thesis.

3.1.2 Related work

There is a number of works on indexing and storing structured documents or XML data related to the issue of data coordinate. Position-based Indexing and Path-based Indexing have been proposed in [33] to indicate the location of data in XML documents. Position-based Indexing contains the positions of words and XML tag names represented by the absolute address. Path-based Indexing encodes all paths leading to each word. NATIX [6] manages tree-structured large objects, preferably XML documents. Data is stored in flat records, which are mapped into logical trees by a tree storage manager. A hybrid model [14] stores XML attributes in a database system, whereas the content of XML elements and their indices are saved in files. Storing XML data using an RDBMS in combination with the flat files have been proposed in XRel [26], where XML data is indexed by using an XML node type scheme and the query result returns the coordinate that is used to retrieve the request data from the original XML documents. Data coordinate also has been used in the system introduced by [13].

XML content updating has not been adequately investigated. The related works mainly focused on the development of query mechanisms and storage schemes. However, the problem attracted more effort when XML became the data format of applications where data is extensively changeable. In [22], a set of instructions has been proposed for expressing the updates using the syntax of XQuery language. Our research investigates the content update robustness in terms of the storage structure and proposes a data structure suitable for the problem. Although XML documents can be decomposed into relational tables, storing XML documents in their entirety is still relevant. For example, this approach has been applied in [26][6] to store large data objects in the flat part of data, which may contain a number of XML elements. Oracle 9*i* [28] and DB2 XML extender of IBM [23] also support the storage of composed XML in CLOB*s*. In many other database systems, LOB (Large OBject) fields are supported to store long character strings [39][1][25]. Using coordinates is the certain choice to address data in these cases, and such designs can benefit from the application of this study.

The content of the chapter is as follows. Section 3.2 introduces the preliminaries of this study. Section 3.3 defines the Relative Region Coordinate. Section 3.4 discusses the application scope of the technique. Section 3.5 presents the general framework for implementing the technique. Section 3.6 discusses an application in detail using the case-study with XRel. Section 3.7 describes the experimental results, and Section 3.8 concludes the chapter with a discussion.

3.2 Region Coordinate of XML Data

For simplicity, let us regard a collection of XML documents as a *concatenated* XML document with an artificial root element, and we will omit the document index in the rest of the chapter. A simple XML document is presented in Example 1.

Example 1 *A simple XML document describes the relationship between Richard Branson, as a manager, and his subordinate employees.*

```
<person>
   <name>
      <family>Branson</family>
      <given>Richard</given>
   </name>
   <email>branson@domain.com</email>
   <person>
      <name>
         <family>Soros</family>
         <given>George</given>
      </name>
   </person>
   <person>
      <name>
         <family>Leeson</family>
         <given>Nick</given>
      </name>
   </person>
</person>
```

The XML tree corresponding to the XML document listed in Example 1 is depicted in Figure 3.1. The root node or an internal node is depicted by a cycle. A text node is marked by its text content.

By *content update* we mean a change in the data stored in a leaf node. It is equivalent to the change of the text between the start-tag and end-tag of the element corresponding to the node. For example, in Example 1, the e-mail address of Branson may be changed from “branson@domain.com” to “bransonrichard@yahoo.com”. The child text node of the corresponding **`email`** node must be changed accordingly.

Since XML elements have changeable lengths, the locations of elements in an XML document cannot be expressed by a fixed scheme. Therefore, we use a region coordinate, which is a pair of integers equal to the absolute distances from the start of the document to the start and the end bytes of the piece of text corresponding to the element. Let us call the address *Absolute Region Coordinate* (*ARC*).

Despite of the convenience of directly locating the data items in an XML document, the integrity of ARC is difficult to maintain. When an update changes the length

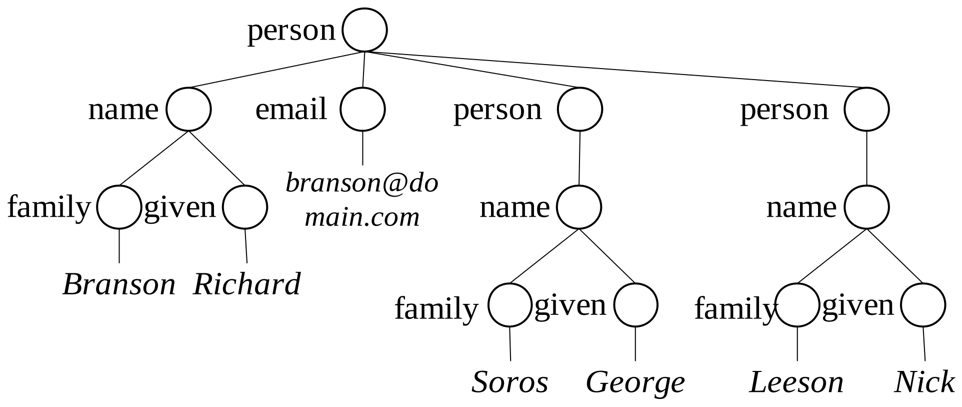


Figure 3.1. XML tree of the XML document in Example 1

of the content of an element, the ARC of the element and of all of its successive elements must be changed. In the XPath terminology, the *ancestor-or-self* nodes and the *following* nodes of the updated node are affected. In general, the ARC of a large number of elements have to be recomputed.

Example 2 *The ARCs of the elements, listed in preorder traverse,* **`person`**, **`name`**, **`family`**, **`given`**, **`email`**, **`person`**, **`name`**, **`family`**, **`given`**, **`person`**, **`name`**, **`family`**, *and* **`given`** *in Example 1 are [0, 253], [8, 66], [14, 37], [38, 59], [67, 99], [100, 172], [108, 163], [114, 135], [136, 156], [173, 244], [181, 235], [187, 209], and [210, 228], respectively. If the content of the element* **`email`** *is changed from "branson@domain.com" to "bransonrichard@yahoo.com", then the data length of the element is increased by 6 bytes. In consequence, the ARCs of these elements become [0, 259], [8, 66], [14, 37], [38, 59], [67, 105], [106, 179], [114, 169], [120, 141], [142, 152], [179, 250], [187, 241], [193, 215], and [216, 234], respectively.*

For simplicity, in this chapter, we assume that the type of an internal node of XML trees in the data model is Element and that of a leaf node is Text.

3.3 Relative Region Coordinate

In this section, we present the Relative Region Coordinate technique. We first present two versions of RRC – the byte version and the word-tag version. We describe the distribution of the nodes, the ARC and RRC of which are affected by an update at a given node. The theoretical evaluation of our technique and its hybrid version are also discussed.

3.3.1 RRC Description – the Byte Version

RRC shows the location, expressed in bytes, of element within the space of its parent element.

Definition 1 *The byte version of Relative Region Coordinate of an element is a pair of integer numbers [r_1, r_2], where r_1 (respectively, r_2) is the number of bytes from the start of the parent element to the start byte (respectively, the end byte) of the element. Values r_1 and r_2 are called the first and the second RRC coordinates.*

Example 3 *The RRCs of the elements, listed in preorder traverse,* **`person`**, **`name`**, **`family`**, **`given`**, **`email`**, **`person`**, **`name`**, **`family`**, **`given`**, **`person`**, **`name`**, **`family`**, *and* **`given`** *in Example 1 are [0, 253], [8, 66], [6, 29], [30, 51], [67, 99], [100, 172], [8, 63], [6, 27], [28, 48], [173, 244], [8, 62], [6, 28], and [29, 49], respectively.*

For convenience, when we refer to the ARC (RRC) of a node in an XML tree, we imply this is the ARC (RRC) of the element corresponding to the node in the rest of the chapter. Given the ARCs of a node and its parent node, the RRC of the child node can be computed. Inversely, the ARC of a node can be computed if the RRCs of its ancestor nodes are all known.

In an XML tree **`T`**, if the RRC of the node $\mathbf{n}_i$ (i = 0, 1,…, k) in the node path $\mathscr{P} \equiv$ ($\mathbf{r} = \mathbf{n}_0$, $\mathbf{n}_1$, $\mathbf{n}_2$,..., $\mathbf{n}_k = \mathbf{n}$) for a node $\mathbf{n}$ is denoted by [a_i, b_i], then the ARC of $\mathbf{n}$ is equal to $[(\sum_{i=0}^{k} a_i), (\sum_{i=0}^{k-1} a_i) + b_k]$.

3.3.2 RRC Description – the Word-Tag Version

In a number of applications, the notion 'word' is of interest rather than 'byte'. A word is considered to be a sequence of characters or digits and bounded by delimitation characters, such as dots, commas, blanks, etc. The word-tag address of a tag is represented by a pair of integers [$w.t$]. In the *absolute* word-tag address [33], w is the number of words preceding the tag in the document. In the *relative* word-tag address, w is the number of words preceding the tag in the region of the parent element of the tag. In both cases, t is the number of open tags preceding the tag, counting from the nearest preceding word. The Word-Tag Region Coordinate of an element is a pair of word-tag addresses of its start and end tags.

Let $\{[w.t], [w'.t']\}$ denote the absolute word-tag coordinate of $\mathbf{n}$, and $\{[w_i.t_i], [w'_i.t'_i]\}$ denote the relative word-tag coordinate of $\mathbf{n}_i$, (i = 0, 1,…, k), in $\mathscr{P}$. Since the w component of the first relative word-tag address of a node equals to the total number of words in the sibling nodes to the left of the node, $w = \sum_{i=0}^{k} w_i$. The equalities $w' = \sum_{i=0}^{k-1} w_i + w'_k$, $t = t_k$, and $t' = t'_k$ are deduced from the definition of relative word-tag address.

Example 4 *In Example 1, the absolute and relative word-tag addresses of the tag* `<given>` *of the element* `<given>George</given>` *are [6.1] and [1.1], respectively.*

Note that the properties described in Sections 3.3.3, 3.3.4, and 3.3.5 are applicable not only to the byte version but also to the word-tag version of RRC.

3.3.3 RRC with Content Update Problem

In this part, we describe how the nodes, ARC and RRC of which are affected by an update at a given node, are distributed in an XML tree.

Definition 2 *For a leaf node* **n***, the nodes, RRC of which have to be updated when an update occurs at* **n***, are called RRC-updated nodes of* **n***. The nodes, ARC of which have to be updated when an update occurs at* **n***, are called ARC-updated nodes of* **n**.

The following observation shows the distribution of the RRC-updated nodes of a node.

Observation 1 *Given a node* **n** *having the node path* $\mathscr{P} \equiv (\mathbf{n}_0, \mathbf{n}_1, \mathbf{n}_2, ..., \mathbf{n}_k)$*, the RRC-updated nodes of* **n** *are* $\mathbf{n}_i$ *and the siblings to the right of* $\mathbf{n}_i$ *(i = 0, 1,…, k). In terms of XPath, the RRC-updated nodes are the ancestor-or-self nodes of* **n** *and the siblings to the right of these nodes, as depicted in Figure 3.2.*

Let us illustrate Observation 1 using the byte version of RRC in the following example.

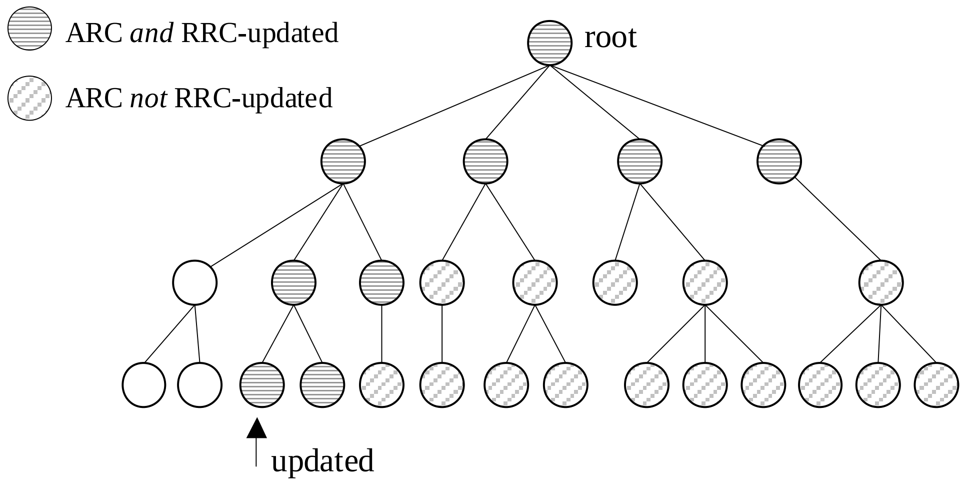


Figure 3.2. ARC and RRC-updated nodes distribution

Example 5 *Using RRC, if the content of the element* **`email`** *is changed from "branson@domain.com" to "bransonrichard@yahoo.com", the RRCs of the only element* **`email`** *and three elements* **`person`** *are changed.*

3.3.4 Theoretical Evaluation of RRC

In this part, we evaluate the effectiveness of the RRC method over ARC in content update. We use the number of updated nodes as an evaluation criterion.

The numbers of ARC- and RRC-updated nodes of a node in an XML tree depend on the shape of the tree and on the position of the node within the tree. Therefore, it is not realistic to evaluate the effectiveness of RRC over ARC in the general tree shape. We choose a simplified context when the XML tree **T** is balanced, all the nodes of the level j (j = 0, 1,..., k-1) have s_{j+1} child nodes, and the data in each leaf node of **T** is changed once.

Hereafter, let rUpd(**n**), rUpd(**T**), aUpd(**n**), and aUpd(**T**) denote the number of RRC-updated nodes of **n**, the total number of RRC-updated nodes of all leaf nodes of **T**, the number of ARC-updated nodes of **n**, and the total number of ARC-updated nodes of all leaf nodes of **T**, respectively.

We numerate the child nodes of each parent node from right to left starting at 1. Given a leaf node **n**, since the node path $\mathscr{P}$ for **n** is represented as (**r**, $\mathbf{n}_1$, $\mathbf{n}_2,\cdots$, $\mathbf{n}_k$ = **n**), the node **n** can be in correspondence with a sequence (1, i_1, $i_2,\cdots$, i_k), where i_j is the order of $\mathbf{n}_j$ (j = 1, 2,$\cdots$, k) among its siblings counting from right to left.

Proposition 1 $\mathit{rUpd}(n) = \sum_{j=1}^{k} i_j + 1$

Proof. Travel from the node **n** upward to the root **r** along $\mathscr{P}$. From Observation 1, among the siblings of $\mathbf{n}_j$ there are i_j nodes, including $\mathbf{n}_j$ and the siblings to the right of $\mathbf{n}_j$, that belong to rUpd(**n**). □

Proposition 2

$$\mathit{rUpd}(T) = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{j=1}^{k} s_j + k + 2) \tag{3.1}$$

Proof. From Proposition 1, the total number of RRC-updated nodes of all leaf nodes of **T** is $\sum_{Leaves\ set}(\sum_{j=1}^{k} i_j + 1)$. We decompose the value into two components: $A = \sum_{Leaves\ set} 1 = \prod_{j=1}^{k} s_j$ and $B = \sum_{Leaves\ set} \sum_{j=1}^{k} i_j$. The sum B is a matrix having k rows, where every value ranging from 1 to s_i appears in the i-th row totally $\prod_{j=1}^{i-1} s_j \times \prod_{j=i+1}^{k} s_j$ or $\prod_{j=1}^{k} s_j / s_i$ times. Therefore, all of the elements corresponding to the i-th row have a sub-sum equal to $\prod_{j \neq i}^{k} s_j(1 + 2 + .. + s_i) = \frac{1}{2}\prod_{j=1}^{k} s_j(s_i + 1)$. Summing up for i, we have $B = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{j=1}^{k} s_j + k)$ or $A + B = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{j=1}^{k} s_j + k + 2)$. □

Now we will evaluate aUpd(**T**). For a given leaf node **n**, let Ord(**n**) denote the order of **n** in the sequence of leaf nodes starting from right to left in the set of all leaf nodes. Since the coordinates are expressed by ARC, any length-update in a leaf node will result in the update of coordinates in not only the parent node but also in all leaf nodes following the node.

Observation 2 *If* **n** *is a leaf node, then aUpd(***n***) is equal to the sum of aUpd(***v***) and Ord(***n***), where* **v** *is the parent node of* **n**.

Let P_j denote the value of aUpd() of the tree constructed from **T** by removing all nodes having the node paths longer than j. According to this notation, P_k is equal to aUpd(**T**).

Proposition 3

$$\mathit{aUpd}(T) = \frac{1}{2}\prod_{j=1}^{k} s_j(\sum_{m=1}^{k}\prod_{j=1}^{m} s_j + k + 2) \tag{3.2}$$

Proof. Each leaf node of P$_{k-1}$ is the parent node of s_k child leaf nodes of P_k. There are $\prod_{j=1}^{k} s_j$ leaf nodes in the tree of P_k. From Observation 2, P_k can be computed as
$P_k = s_k P_{k-1} + \frac{1}{2}(\prod_{j=1}^{k} s_j + 1)\prod_{j=1}^{k} s_j$

Recursively apply the formula for P$_{k-1}$ and so on:

$P_k = s_k s_{k-1} P_{k-2} + s_k \frac{1}{2}(\prod_{j=1}^{k-1} s_j + 1)\prod_{j=1}^{k-1} s_j$
$\quad + \frac{1}{2}(\prod_{j=1}^{k} s_j + 1)\prod_{j=1}^{k} s_j$
$P_k = s_k s_{k-1} P_{k-2} + \frac{1}{2}(\prod_{j=1}^{k-1} s_j + 1)\prod_{j=1}^{k} s_j$
$\quad + \frac{1}{2}(\prod_{j=1}^{k} s_j + 1)\prod_{j=1}^{k} s_j$
...
$P_k = \prod_{j=1}^{k} s_j P_0 + \frac{1}{2}\prod_{j=1}^{k} s_j(\prod_{j=1}^{k} s_j + \prod_{j=1}^{k-1} s_j + ... + s_1 + k)$

Since P_0 = 1 and aUpd(**T**) = P$_k$, the proposition holds. □

From Equations (3.1) and (3.2), the update node reduction ratio of RRC to ARC is equal to:

$$\frac{\sum_{m=1}^{k}\prod_{j=1}^{m} s_j + k + 2}{\sum_{j=1}^{k} s_j + k + 2}$$

In practice, this ratio is greater than one when k and most of s$_i$ are equal to or greater than two. Figure 3.3 depicts the increasing trend of the ratio in the exponential scale.

3.3.5 The ARC-RRC Hybrid Technique

ARC and RRC can be used in hybrid mode. Given an XML tree, we select a set of *representative* nodes, denoted by *r-node*, as illustrated in Figure 3.4(a). In the hybrid mode, the RRC of a node is computed in the region of the node's lowest ancestor r-node, instead of the parent node. The hybrid technique reduces the scope of the content update as RRC does. Moreover, the computation of the ARC of a node involves only the r-nodes, bypassing other intermediate nodes, in its node path. For example, in Figure 3.4(b), the computation of the ARC of **n** involves **r**$_1$ and **r**$_2$, the only r-nodes in the node path from the root **r** for **n**.

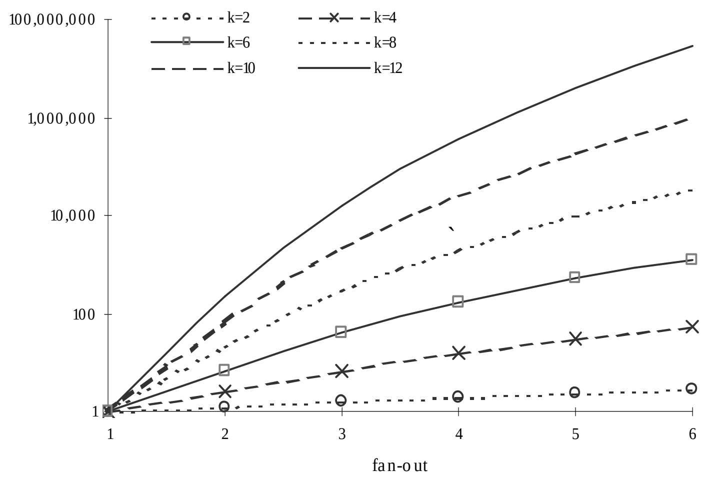


Figure 3.3. Update node reduction ratio of RRC to ARC

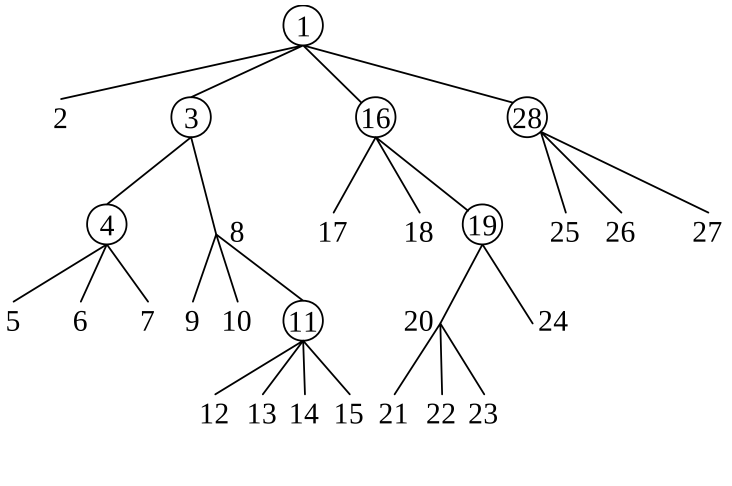
The set of r-nodes can be adjusted as following: “If the query frequency is high or the update frequency is low, then ARC should applied. The r-node of this part should be *promoted* to be a node near to the root of the XML tree. Otherwise, the r-node of this part should be *demoted*.”

3.4 Applications of RRC

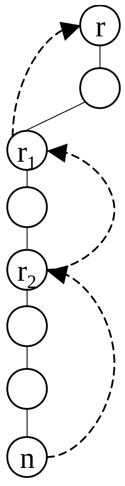
The theoretical evaluation of the effectiveness of RRC presented in Section 3.3.4 does not depend on any specific implementation. This theoretical result shows the superiority of RRC over ARC as a general tendency. However, for the following reasons, this result by itself is not sufficient to demonstrate the usefulness of RRC in real XML database systems:

- The result in Section 3.3.4 is based on a completely balanced XML tree.
- Many aspects affect the performance in real XML database systems.

Also, we need to verify that query processing time is not sacrificed for the sake of efficient update processing. Therefore, it is necessary to implement the concept of



(a) representative nodes



(b) RRC dependence

Figure 3.4. Hybrid technique and its advantage

RRC in real XML database systems and to evaluate the performance of query and update processing using practical XML data.

There are several possible approaches to the implementation of RRC in XML storage or indexing. A fundamental approach is developing a native XML index structure based on RRC. In [9], we developed a tree index structure in which XML nodes, updated together at high probability, are clustered together. However, performance was not evaluated in [9]. Another important approach is storing a string representation of the XML document and then representing the position of XML nodes by the RRC values. In principle, the idea of RRC is applicable to any method employing ARC. The focus of the rest of the chapter is to demonstrate the usefulness of RRC in the latter approach.

3.5 Tree Structural Coding and Query Processing Framework for RRC

3.5.1 Tree Structural Coding

RRC shows the position of an element in the region of its parent node, not in the entire XML document. To identify the elements in an XML document, besides RRC we need additional information about the structure of the XML tree, called *tree structural coding*. The coding is also used to compute ARC from RRC, hence it is desirable that coding have a small size.

There are simple tree codings such as the *Ancestor-Listing* and *Parent-Listing*. However, these codings cannot present ordered trees. In this research, we adopt the *orders of the start and end tags* coding, which assigns a pair of numbers [`pre`, `post`] to every XML element where `pre` and `post` satisfy the *region containment* condition: if a node **p** is the parent node of the node **q** then $\mathtt{pre}_p < \mathtt{pre}_q \leq \mathtt{post}_q < \mathtt{post}_p$. We also consider variants of the coding as follows:

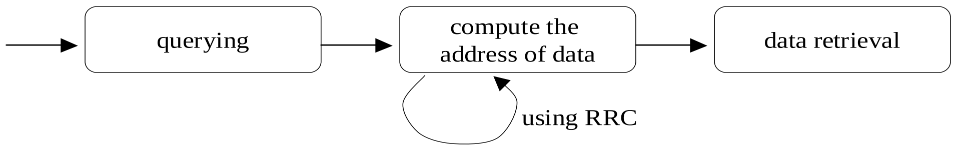
1. **Sparse node enumeration**: The nodes are enumerated in preorder and postorder traversals in an XML tree. However, the enumerations are generated in the sparse mode, hence the coding is robust in the structural change to the XML document, e.g. a node insertion.

2. **Preorder and rightmost leaf**: In an XML tree, if the nodes are enumerated in preorder, the *righmost* leaf of a node **p** is a leaf node, i) which is **p** itself or a descendant of **p**; and ii) which has the largest preorder identifier among the leaf nodes satisfying the condition i). For a given node **p**, let $\mathtt{pre}_p$ be the preorder of **p**, and let $\mathtt{post}_p$ be the preorder of the rightmost leaf of **p**. Then, for a child node **q** of **p**, the inequalities $\mathtt{pre}_p < \mathtt{pre}_q \leq \mathtt{post}_q \leq \mathtt{post}_p$ holds.

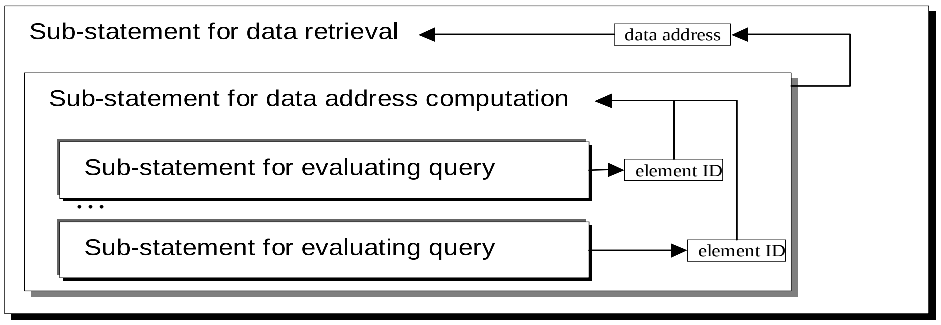
To make the coding convenient to determine the hierarchical level, we extend it by adding the *node level* information, i.e. [`pre`, `post`, `nlevel`]. The introduction of structure coding enables the computation of ARC from RRC.

3.5.2 Query Processing Framework with RRC

Query processing with RRC has three steps, as shown in Figure 3.5(a). The structure of the program code is illustrated in Figure 3.5(b). First, element IDs qualifying the query condition is determined. Then, their ARCs are computed from RRC values. Finally, the result data is retrieved from the text representation of the XML document using the ARC values.



(a) Query processing steps


Sub-statement for data retrieval
data address
Sub-statement for data address computation
Sub-statement for evaluating query
element ID
...
Sub-statement for evaluating query
element ID

(b) Structure of program code

Figure 3.5. The query processing framework

Querying on structure with XML queries is performed using the tree structure coding without the involvement of the element coordinates.

3.6 An Adaptation Approach: Storing XML Data with RRC using an RDBMS

There have been many methods for storing XML data proposed so far[3]. Although the idea of RRC is generally applicable to any XML database system that employs ARC, it is not feasible to implement RRC on every such database systems to verify the effectiveness of RRC. One of the important approaches to exploiting the RRC technique is to integrate RRC in XML databases using an RDBMS. In this section, we describe how RRC can be incorporated into XRel[26], an XML database system built on top of RDBMSs. We have chosen XRel as a testbed because the system was developed by the authors' group, hence it is easy to modify the source code and implement the concept of RRC.

3.6.1 The Basic Scheme

In XRel, a relation is created for each node type `Element`, `Attribute`, and `Text`. Each instance of a node is stored by a tuple in the corresponding relation. The fourth relation stores the list of enumerated path expressions. A path expression in this relation is a concatenation of element names in a path from the root node to a node in an XML tree. The fifth relation stores entire XML documents using LOB, the Large Object type of the database attribute. The schemas for these relations are as follows.

ELEMENT (`did, pid, start, end, ind, reind`)
ATTRIBUTE (`did, pid, start, end, avalue`)
TEXT (`did, pid, start, end, tvalue`)
PATH (`pid, pathexp`)
DOCUMENT (`did, dvalue`)

where the database attributes `did`, `pid`, `start`, `end`, `ind`, `reind`, `avalue`, `tvalue`, `pathexp`, and `dvalue` represent a document identifier, a path expression identifier, the start and end positions of a region, the occurrence order of an element among its siblings and that in reverse order, an attribute string value, a text value, a path expression, and the content of a document, respectively.

3.6.2 RRC-supported Scheme

To apply RRC, the basic scheme is transformed as follows: (1) each element, attribute and text node is stored with its `pre` and `post` from the tree structural coding, (2) the ARC `start` and `end` are removed from the relations ELEMENT, ATTRIBUTE and TEXT, and (3) a new relation STRURRC stores RRC, `pre`, `post`, and `nlevel` of all element, attribute and text nodes. The new RRC-supported relational scheme, denoted by RSS, is the following:

ELEMENT (`did, pid, pre, post, ind, reind`)
ATTRIBUTE (`did, pid, pre, post, avalue`)
TEXT (`did, pid, pre, post, tvalue`)
PATH (`pid, pathexp`)
DOCUMENT (`did, dvalue`)
STRURRC (`did, pre, post, sr, er, nlevel`)

where the database attributes `pre`, `post`, and `nlevel` are the components of the tree structural coding described in Section 3.5.1 and `sr` and `er` are components of RRC. Other attributes are the same as those in the basic scheme.

3.6.3 SQL Statements for RSS

In RSS, the SQL statements have to be transformed to incorporate the RRC values. The query processing has three steps, as shown in Figure 3.5(a), and SQL statements of RSS have a three-loop structure in correspondence to the steps in Figure 3.5(b).

In XRel, the ARC values `start` and `end` are used to determine the hierarchical order of nodes and to address the nodes. In the RSS scheme, the hierarchical order is established by the tree structural coding while RRC is only used to compute the address of data. We replace the parts of the SQL statement on structure that are expressed using ARC in XRel by rewriting them based on the tree structural coding. The replacement results in a new SQL statement that is equivalent to the original one of XRel in terms of the query output. We summarize the main transformation steps for XPath axes as follows. Note that the list below covers all major XPath axes.

- `ancestor-descendant`: In XRel, the relationship is guaranteed by the condition "`a.start < b.start AND a.end > b.end`". In RSS, the corresponding condition is "`a.pre < b.pre AND b.post < a.post`".

  For `ancestor-or-self` and `descendant-or-self` axes, the inequality $<$ in the above condition is replaced by $\leq$.

- `parent-child`: This can be determined from `ancestor-descendant` relationship and `nlevel`. The `nlevel` of the parent element is equal to the `nlevel` of the child element minus one.

- `following-preceding`: In XRel, the relationship is guaranteed by the condition "`a.start < b.start AND a.end < b.end`". In RSS, the corresponding condition is "`a.pre < b.pre AND a.post < b.post`".
- `sibling`: Two nodes in a tree are siblings if they have the same parent node. The `preceding` and `following` orders of the siblings, i.e. `preceding-sibling` and `following-sibling` axes, are determined based on their `pre` and `post` values.

In the new scheme we separated the data used for querying on structure (using the structural presentation) and the data used for the content retrieval. The ARC of a node is computed based on the RRC of all of its ancestor-or-self nodes. In SQL, the ARC of an element identified by [*vPre*, *vPost*] is computed based on the table STRURRC such as:

```
SELECT sum(p.sr), sum(p.sr) + min(p.er-p.sr)
FROM ppl p
WHERE p.pre <= vPre AND vPost <= p.post
```

The table STRURRC is used every time the actual address of data is computed.

3.7 Experiment

In this section, we present the results of our experiment to evaluate the effect of RRC on query processing and content update for XML data.

3.7.1 Experimental Platform

The experiments were conducted on a workstation running Windows XP Professional with two 2-GHz CPUs, 2 GB of RAM, and a hard disk. The parser for XML data is the DOM/SAX parsers available from the Xerces project [4]. Other programs were written in Java. We used several modules borrowed from [26] to transform the XML data into a relational database.

Parameter	Value
Size of data	5700KB
Number of elements	83533
Number of attributes	19249
Text size	3935KB

Table 3.1. Specification of data set

did	pre	post	sr	er	nlevel
1	40	5873330	40	5873338	0
1	45	2909100	8	2909066	1
1	50	98350	11	98309	2
1	60	1720	10	1661	3
1	80	120	19	52	4
1	120	140	55	76	4
1	140	180	79	113	4

Table 3.2. Instance of the STRURRC table

To create the data set for our experiments, we used the freeware provided by XMark[2] to generate a synthetic XML document based on the DTD “auction.dtd”. The specification of the data set is shown in Table 3.1. A portion of an instance of the table STRURRC is shown in Table 3.2.

3.7.2 Queries of ARC and RRC

The goal of this experiment is to compare the time for query processing of the basic scheme of XRel and the new RSS scheme. We utilize a sample query set shown in Table 3.3, which contains various kinds of queries: a simple path expression, a long path expression, a path expression with one ancestor-descendant relationship (one ’//’), a path expression with one ancestor-descendant relationship and one parent-child relationship (one ’//’ and one ’/’), and a path expression with two ancestor-descendant relationship (two '//') with text matching. To avoid the effect of buffering, we run each test several times and compute the average of results as the time for the test. The SQL statements of the queries are written using the structure depicted in Figure 3.5(a)(b). As an example, the SQL statement for Query 1 is shown in Figure 3.6.

Queries	**XPath expressions**
Q_1	`/site/regions`
Q_2	`/site/categories/category/description/parlist/listitem/text`
Q_3	`//edge`
Q_4	`//person/profile[education = ‘Graduate School’]`
Q_5	`//people//person[creditcard = ‘5400 1632 3033 7747’]`

Table 3.3. Test query set for RRC on XRel

The result of this test is shown in Table 3.4, where the columns **Queries**, **XRel**, **RSS**, **RSS/20**, **Output** represent the query index, the query times for the query in XRel, in RSS, in RSS if the number of output elements is limited by 20, and the actual number of output elements, respectively.

Queries	**XRel**	**RSS**	**RSS/20**	**Output**
Q_1	1305	1150	-	1
Q_2	578	875	820	32
Q_3	3001	3563	1406	50
Q_4	10019	10082	5019	85
Q_5	19804	6445	-	1

Table 3.4. Query time in XRel and RSS in *msec*

Figure 3.7 shows the graphical comparison of query processing time in XRel and RSS. From Figure 3.7, we can observe that the query processing time in RSS is comparable with that in XRel. It is worth noting that RSS performs better than XRel for pro-

```
SELECT substr(d.dvalue, c.x, c.y-1)
FROM document d,
    (SELECT sum(p.sr) AS x, min(p.er-p.sr) AS y, p.did
     FROM (SELECT p1.sr, p1.er, p1.did, p2.pre
         FROM ppl p1,
             (SELECT e0.did, e0.pre, e0.post
              FROM element e0, path p0
              WHERE p0.pathexp LIKE '#/site#/regions'
              AND e0.pid = p0.pid) p2
         WHERE p1.pre <= p2.pre AND p2.post <= p1.post
         AND p1.did = p2.did) p
    GROUP BY p.did, p.pre) c
WHERE d.did = c.did
```

Figure 3.6. SQL statement of Query 1 for RRC on XRel

cessing some queries. A part of the extra time is paid for the interconnection between Java and RDBMS and to compute the actual address of data. The query processing in RSS can be done either entirely in one-statement-SQL mode or in an interactive-SQL-Java mode. In the latter mode, we first find the identifiers of the result data and the documents that contain the data, then use the table STRURRC to compute the actual address of the data, and finally load the data from the documents.

We also consider variants of the implementation:

Pre-compute ARC. If we can anticipate a heavy query load, the large cardinalities of query results, and the areas in the XML documents where the results will be located, then the ARC value can be computed from the RRC in advance.

Threshold for the number of output. We set a threshold for the cardinality of the output. For example, in processing the query Q_3, if the threshold is set to be 20, then the query time is only 1406. The setting can be applied when the users seek only the most relevant output of queries, not for listing purposes.

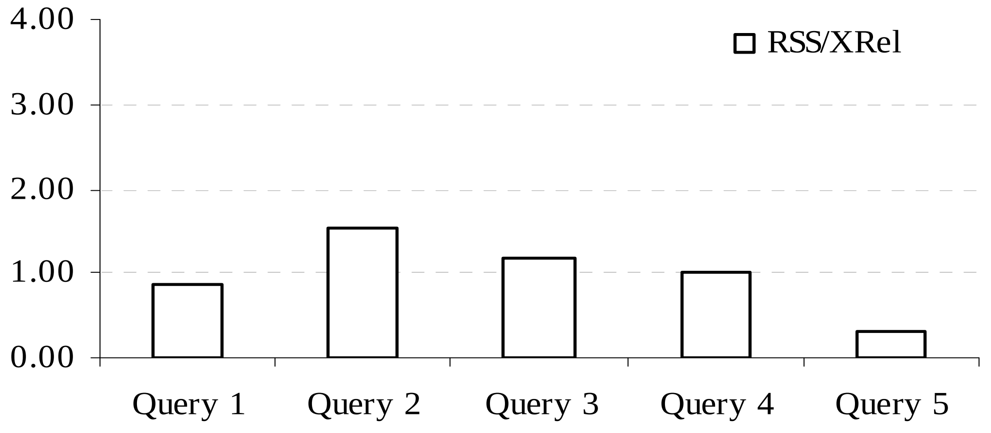


Figure 3.7. Ratio of query times of RSS to XRel

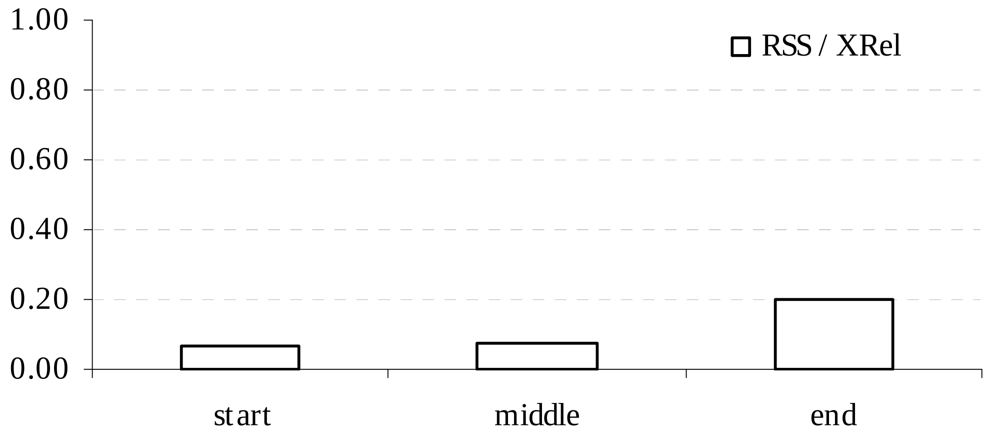


Figure 3.8. Ratio of update times of RSS and XRel

3.7.3 Update of RRC and ARC

We performed several simple content updates using ARC and RSS configurations. Specifically, we selected a sample set of three nodes and performed a content update in these nodes. The test nodes are distributed in different parts of the XML data: at the beginning, the middle and the end of the data file. Each content update operation is stated as follows: "Change the content of a text node identified by [*pre*, *post*] from $string_1$ to $string_2$, where l = len($string_2$) - len($string_1$), in the document having the identifier equal to *did*".

In XRel and RSS, the content update operation consists of the following steps (grouped by the tables accessed):

In both schemes: 1) update the `dvalue` column of the corresponding tuple in the table DOCUMENT

In XRel only: 2) update `pre` and `post` in ELEMENT, 3) update `tvalue`, `pre`, and `post` in TEXT, and 4) update `pre`, `post` in ATTRIBUTE

In RSS only: 3.1) update `tvalue` in TEXT, and 5) update `sr` and `er` in STRURRC

The results of this test are illustrated in Figure 3.8. The update time in RSS is 5 to 10 times faster than in XRel. Despite the dependence on the internal system architecture and the data structure, the RRC application still provides efficiency in content update processing. It is worth noting that the total of the data loading times for the database in both schemes is much longer than the update time. Therefore, the possibility of a simple and naive approach for updating that performs total data reconstruction is eliminated.

3.8 Summary of Chapter 3

In this chapter, we investigated the problem of how to maintain the coordinate integrity for XML data in the content update. This function is important in XML information systems, where data is changed frequently. We found that expressing the location of XML elements by the absolute region coordinates causes a heavy workload for the reconstruction of index files when a content update occurs. We proposed a new method, called Relative Region Coordinate, that deals effectively with this problem. In our method, the coordinate of an XML element is expressed based on the region of its parent element or an appropriately selected ancestor element. Consequently, if an update occurs, the workload for recomputation of the coordinates is reduced. We demonstrated a procedure to integrate the Relative Region Coordinate technique into XML systems to enhance their robustness in content update through a case-study with XRel. The experimental result showed that the Relative Region Coordinate technique can significantly improve the content update performance of the target system. This technique is applicable to any XML systems that employ the character string representation for XML data.

Chapter 4

A Structural Numbering Scheme for Processing Queries by Structure and Keyword on XML Data

4.1 Introduction

In Chapter 3, we presented a technique for making XML indexing techniques more robust on the content update problem of XML data. In this chapter we are going to investigate another problem that is related to the hierarchical feature of XML data and its structural update.

The main components of an XML document are elements, which can have extra information attached to them called attributes. XML elements may have various lengths and locate in a hierarchy that is the structure of the XML document. The structure of an XML document may change when elements are inserted or deleted. Therefore, a presentation of the structure of XML documents is essential for processing the queries on these documents. A number of studies have discussed the methods to present the structure of XML data in a concise manner. An effective presentation must be compact such that it requires small amount of I/O and CPU workload to be manipulated.

In addition, in order to distinguish an XML element or attribute from others, unique identifiers are assigned to them as the key used in the relational data model. Generating identifiers for the XML elements and attributes is a common but crucial task in XML applications. The method of generating identifiers normally determines the structure of indexes as well as the query processing mechanism.

A number of numbering schemes for XML data have been proposed in previous stuties [52][31][34][20] that meet two above requirements: generating the identifiers of XML elements and expressing the structural information. Normally, a numbering scheme assigns identifiers to elements such that the hierarchical orders of the elements can be re-established based on their identifiers. Since hierarchical orders are used extensively in processing XML queries, the reduction of the computing workload for the hierarchy re-establishment is desirable.

4.1.1 Contribution

In this chapter, we present a novel numbering scheme called the *recursive UID* (*r*UID) [10][11] that is an extended version of the UID technique [52]. The basic idea of *r*UID, which is represented in Section 4.2 through 2-level UID and its general version, is that the nodes of an XML tree are enumerated in a number of levels. In each level, the nodes are clustered and represented by a combination of an area number and the node number inside the area. The main features of the *r*UID technique are:

1. *Parent-child relationship determination:* given the identifier of a node, the parent node's identifier can be efficiently computed. Using small-size global information stored in main memory, *r*UID technique allows the ancestor-descendant relationship to be determined without any I/O.

2. *Robustness for structural change:* the scope of element identifier amendment when a structural update occurs is effectively reduced.

3. *Scalability:* *r*UID can overcome the identifier limitation of the original UID technique and can be applied to large XML documents.

*r*UID is efficient in representing the main axes of XPath expressions. In Section 4.3, we describe an implementation of *r*UID in SKEYRUS (*S*tructure and *KEY*word search based on *R*ecursive *U*id *S*ystem), which integrates structure and keyword searches on XML data. The input of SKEYRUS is a simplified XPath expression with word-containment predicates. We have developed a structural join mechanism for *r*UID to process structural part of XPath expressions. Structural joins select the pairs of XML elements or attributes from the candidate sets such that a given hierarchical order holds. The experimental results presented in Section 4.4 using datasets of various sizes have shown the effectiveness of SKEYRUS in processing the queries on both XML structure and keywords.

This research's result can be applied in the applications categorized into the *Native XML Databases* and *Content Management Systems* groups, as discussed in Section 2.3.1, page 23 of this thesis.

4.1.2 Related work

Due to the hierarchical structure of XML data, structural joins for determining the ancestor-descendant relationship is essential. The approach to determine the relationship using *preorder* and *postorder* has been introduced in [30]. Extensions of the method by using the *preorder* and *range* or *position* and *depth* have been presented in [31][37][8]. The issue of minimizing the size of the node labeling for a tree has been addressed in [34][20]. The prefix-based interval labeling for general trees has been introduced in [34], whereas the prefix-base bit string labeling suitable to XML documents has been presented in [20]. The proposed labeling techniques support the ancestor-descendant relationship determination.

Although independently developed, it is interesting that our approach, which has been originally introduced in [10] and [11], to assign identifiers to nodes in levels is similar to the approach adopted in [34]. The space-efficiency is not the primary goal of the design of *r*UID. However, our solution supports not only the ancestor-descendant but also the parent-child relationship that is essential in XML queries. In addition, the ability to compute the identifiers of the parent and ancestor nodes of a child node accelerates processing the structural joins. It is not clear which join mechanisms are suitable for the proposals in [34] and [20]. The manipulation of our numbering scheme can be done by simple arithmetic operations supported by all of RDBMSs.

Our proposal is spirited by the UID technique, another type of numbering scheme, applications of which have been described in [52][18][14]. The technique enumerates nodes of an XML tree using a k-ary tree where k is the maximal fan-out of the nodes. Each of internal nodes supposedly has the same fan-out k by assigning a number of virtual children if needed. Consecutive integers starting from 1 are assigned to the nodes, including the virtual nodes, in order from top to bottom and from left to right in each level. In Figure 4.1(a), k is equal to 3, the root node is enumerated as 1 and the nodes 4, 10, 11, 12, and 13 are virtual nodes. Whereas other numbering schemes only manages to compare two already known identifiers to determine the parent-child relationship, the UID has an interesting property whereby the parent node identifier can be computed using the formula:

$$\textbf{parent}(i) = \lfloor (i-2)/k+1 \rfloor \tag{4.1}$$

where i is the identifier of the child node.

However, in the previous studies related to the UID technique, the problems of structural update and the capacity of the numbering scheme have not been discussed. When a new node is inserted, the identifiers of all sibling nodes to the right of the inserted node are increased by one. In addition, the identifiers of the descendant nodes of the moved sibling nodes also will be changed. If the number of children nodes of a node becomes larger than the pre-defined value k, then there is no space for a new child node. The modification of k results in an overhaul of the identifier system in which the identifiers of all nodes are to be recomputed. For example, if a node is inserted between nodes 2 and 3 in Figure 4.1(a), then the previous nodes 3, 8, 9, 23, 26 and 27 are re-numerated to be the nodes 4, 11, 12, 32, 35, and 36, respectively, as shown in Figure 4.1(b). If another node is inserted behind the node 4 in Figure 4.1(b), then the entire tree must be re-numerated.

In addition, the original UID enumerates not only the real nodes but also the virtual nodes of a tree. For an XML tree with the maximal fanout m and the depth d, the required number of identifiers are $O(m^d)$. Therefore, in practice, additional purpose-specific libraries are required to deal with the identifiers that exceed the maximal integer value. As illustrated in Section 4.4, the original UID technique even fails to numerate entirely a sample unbalanced XML document of 7.6KB.

The issues of integrating keyword search into XML query processing was addressed in [13], where the evaluation of structural part of a query is performed by joining tables, into which XML data is decomposed. In our approach, we use our numbering scheme to do the evaluation task.

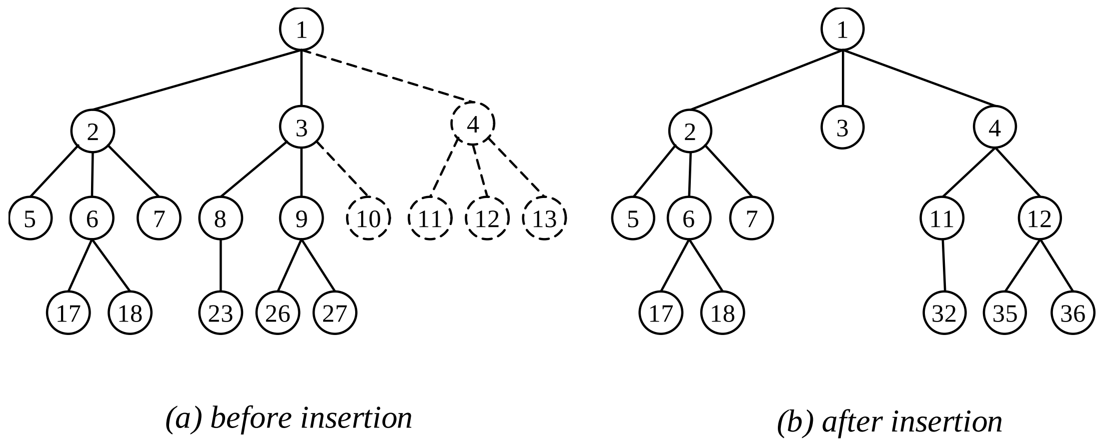


Figure 4.1. Original UID before and after a node insertion.

4.2 Recursive UID

In this section, we describe the recursive UID numbering scheme using the notation of XML tree. An XML document can be expressed by an ordered tree. For example, Document Object Model [46] presents an XML document as a hierarchy of Node objects of different node types. For the purpose of the study, we only consider the element and attribute nodes. The *r*UID technique has been designed to generate identifiers for the nodes of such ordered XML trees. Hereafter, the terminologies '*element*' and '*element node*', '*attribute*' and '*attribute node*', are used interchangeably.

The basic idea of *r*UID is that *r*UID clusters the nodes of an XML tree into connected subgraphs, and represents the nodes by a combination of an area number and node number inside the area, thereby it overcomes the drawback of the original UID technique. If the size of a subgraph is still too large, this technique can be applied recursively; i.e. an area can be further decomposed into subareas. We call the *r*UID technique recursively applied *n* times, *n*-level *r*UID.

In the rest of this section, we first introduce 2-level *r*UID, then present more general *n*-level *r*UID. Finally, we present features of the *r*UID technique.

4.2.1 Description of 2-level *r*UID

The 2-level *r*UID assigns the identifiers to the nodes of an XML tree in two levels. Firstly, the nodes of the XML tree are partitioned into areas called *UID-local area*. Next, the newly created areas are enumerated by the *global* indexes. Finally, the nodes of each area are enumerated by the *local* indexes.

We first present a definition of *frame nodes* before presenting a formal definition of UID-local areas. Frame nodes are a collection of *representative* nodes, each of which represents the nodes in an UID-local area. In other words, frame nodes play a role for bounding UID-local areas.

Definition 3 *(frame nodes) Given an XML tree $\mathscr{T}$ rooted at* $\mathbf{r}$*, a set of frame nodes $N_{\mathscr{F}}$ of $\mathscr{T}$ is a subset of the nodes of $\mathscr{T}$ such that* $\mathbf{r} \in N_{\mathscr{F}}$.

For example, in Figure 4.2(a), $\{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}\}$ is a set of frame nodes of the entire tree.

Next we give a formal definition of UID-local area. A connected subgraph of a tree can be regarded as a smaller tree. We will use terminologies of trees (such as root, internal node and leaf) also for a connected subgraph of a tree.

Definition 4 *(UID-local area) Let $\mathscr{T}$ be an XML tree. Also, let $N_{\mathscr{F}}$ be a set of frame nodes of $\mathscr{T}$. The UID-local area $\mathscr{L}_n$ of* $\mathbf{n}$ *($\in N_{\mathscr{F}}$) is a connected subgraph of $\mathscr{T}$ such that:*

1. $\mathbf{n}$ *is the root of $\mathscr{L}_n$;*
2. *the internal nodes of $\mathscr{L}_n$ other than* $\mathbf{n}$ *are not members of $N_{\mathscr{F}}$; and*
3. *the leaves of $\mathscr{L}_n$ are leaf nodes of $\mathscr{T}$ or members of $N_{\mathscr{F}}$.*

Proposition 4 *For a set of frame nodes $N_{\mathscr{F}}$ of $\mathscr{T}$, and a node* $\mathbf{n}$ *($\in N_{\mathscr{F}}$), the UID-local area of* $\mathbf{n}$ *is uniquely obtained.*

The tree in Figure 4.2(b) is a UID-local area of **c** of the tree in Figure 4.2(a). It is easy to see that an XML tree is covered by UID-local areas such that two areas have at most one common frame node.

Next, we define *frame* which is a tree consisting of frame nodes. A frame is used to numerate UID-local areas.

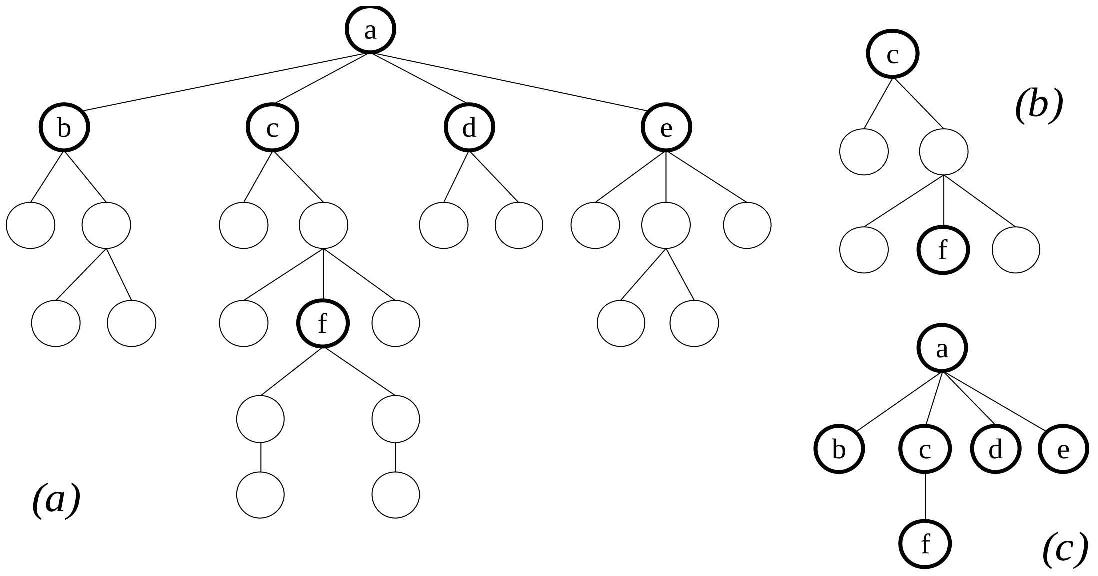


Figure 4.2. Frame and UID-local area.

Definition 5 *(A frame) Given an XML tree $\mathscr{T}$ rooted at* **r** *and a set of frame nodes $N_{\mathscr{F}}$, a frame $\mathscr{F}$ is a tree such that: (1) the root node is* **r***, (2) the node set of $\mathscr{F}$ is $N_{\mathscr{F}}$; and (3) for any two nodes* **u** *and* **v** *in $\mathscr{F}$, an edge (***u***,* **v***) exists in $\mathscr{F}$ iff no frame node (other than* **u** *and* **v***) appears in the path (***u***,* **v***) in $\mathscr{T}$.*

For example, the tree shown in Figure 4.2(c) with the nodes **a**, **b**, **c**, **d**, **e**, and **f** is a frame of the tree in Figure 4.2(a). There exists no edge connecting the node **a** and **f** because the node **c**, which belongs to $\mathscr{F}$, lies between **a** and **f** in $\mathscr{T}$.

Let κ denote the maximal fan-out of nodes in a frame $\mathscr{F}$. We use a κ-ary tree to enumerate the nodes of $\mathscr{F}$ and let the number assigned to each node in $\mathscr{F}$ be the index of the UID-local area rooted at the node. The 2-level UID based on $\mathscr{F}$ is defined as follows:

Definition 6 *(2-level rUID) The 2-level rUID of a node* **n** *is a triple $<g, l, r>$, where g, l, and r are called the global index, local index, and root indicator, respectively. If* **n** *is a non-root node of an UID-local area, then g is the index of the UID-local area containing* **n***, l is the index of* **n** *inside the area, and r is $false$. If* **n** *is the root node* *of an UID-local area, then g is the index of the area, l is the index of* $\mathbf{n}$ *as a leaf node in the upper UID-local area, and r is $true$. The identifier of the root of the main XML tree is <1, 1, $true$>.*

The *global parameters*, which are loaded into the main memory during query processing, consists of κ and the table $\mathscr{K}$, each row of which corresponds to an UID-local area and contains the *global index* of the area, the *local index* of the root of the area in the upper area, and the maximal *fan-out* of nodes in the area.

Note that 2-level *r*UID is a clean, but not straightforward generalization of the original UID technique. The key technique of 2-level *r*UID is the introduction of the enumeration of root nodes of a UID-local area. By using frames and UID-local areas, *r*UID reduces the number of virtual child nodes since subtrees are enumerated by the local fan-outs.

Algorithm 1: Computation of 2-level *r*UID

Input: An XML tree $\mathscr{T}$

Output: The 2-level *r*UID identifiers of nodes in $\mathscr{T}$

```
    //Global enumeration
1.  Partition $\mathscr{T}$ into UID-local areas
    and build the frame $\mathscr{F}$ upon the roots
2.  Find the maximal fan-out $\kappa$ of $\mathscr{F}$
3.  Compute the global index $g_i$ of $\mathscr{F}$
    //Local enumerations
4.  foreach $i^{th}$ UID-local area
5.    find the local maximal fan-out denoted by $k_i$
6.    compute the local indices $l_{ij}$
      of nodes in the area via a $k_i$-ary tree
7.    if $l_{ij}$ = 1 then
8.      recompute $l_{ij}$ in the upper UID-local area
9.      $r_{ij}$ := $true$
10.     update $\mathscr{K}$ using ($g_i$, $l_{ij}$, $k_i$)
11.   else $r_{ij}$ := $false$ endif
12.   Generate $r$UID of the nodes from ($g_i$, $l_{ij}$, $r_{ij}$)
13. end
e.  Save $\kappa$ and $\mathscr{K}$
```

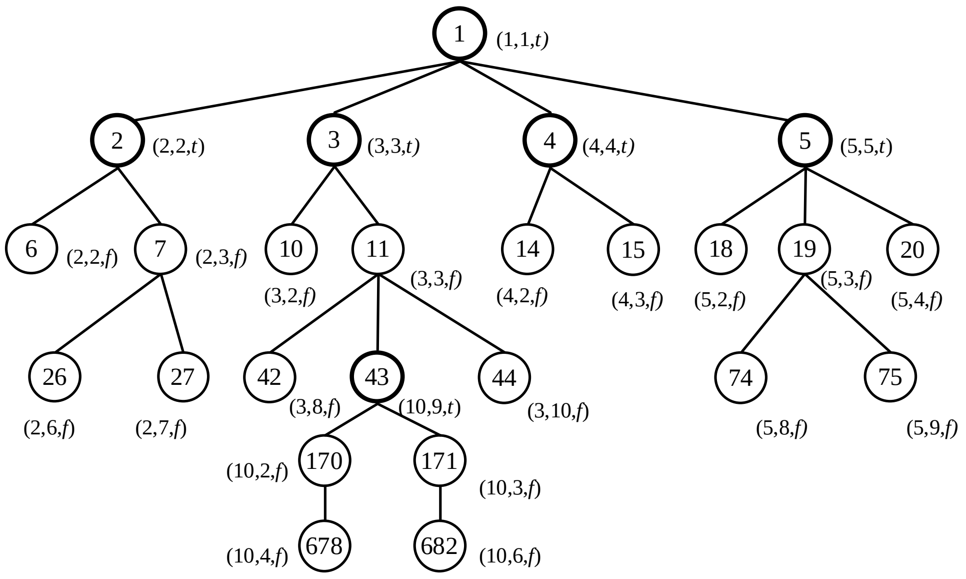


Figure 4.3. Original UID and its 2-level *r*UID counterpart.

The numbering for *r*UIDs depends on the selected frame. The *r*UID of a node in an XML tree is determined uniquely if the frame is fixed. The computation of 2-level *r*UID is briefly described in Algorithm 1. In the step 1 of the algorithm, an area can become an UID-local area if the number of nodes, including the virtual ones, in the area does not exceed a high threshold. In the frame construction, there is a trade-off between the robustness in structural update and the efficiency in processing structural joins. The small size of the UID-local areas will narrow the scope of the identifier change when a new node is inserted. However, since the XML tree is divided into small areas, the size of the table $\mathscr{K}$, which is proportional to the the number of such areas, is larger. In the experiments presented in Section 4.4, the high threshold of the size of an UID-local area is the maximal value of an integer.

Example 6 *Figure 4.3 depicts the original UID, shown inside each node, and the 2-level rUID, as the triple shown next to each node, of a tree. The frame consists of 6 nodes 1, 2, 3, 4, 5, and 43 encircled by bold circles. The UID-local area rooted at the node 3 has the nodes 3, 10, 11, 42, 43, and 44. The global fan-out κ is 4 and the table $\mathscr{K}$ is shown in Figure 4.1.*

Global index	Local index	Local fan-out
1	1	4
2	2	2
3	3	3
4	4	2
5	5	3
10	9	2

Table 4.1. The table $\mathscr{K}$ for the 2-level *r*UID in Figure 4.3.

For some UID-local areas, the local maximal fan-outs are equal to the maximum fan-out of the original XML tree. However, in contrast to the UID method where the identifiers of nodes increase exponentially to the depth of the XML tree, using *r*UID, we can restrict the depth of such local areas.

4.2.2 Parent-Child Relationship in 2-level *r*UID

Formula (4.1) is used to check if a node is the parent node of another node by their UIDs. To determine the parent's 2-level *r*UID of a node, we need a more sophisticated function shown in Algorithm 2. Suppose that an XML tree $\mathscr{T}$, a node **n** in $\mathscr{T}$, and the parent node **p** of **n** are given. If **n** is the root of an UID-local area, then the global index of **p** is computed from the global indices of **n** using the formula (4.1). Otherwise, the global indices of **n** and **p** are equal (steps 1-4). Knowing the UID-local area that **p** belongs to, the maximal fan-out of the area can be found from the table $\mathscr{K}$ (step 5). Initially, the local index of **p** is computed from the local index of **n** (step 6). If the result is equal to 1, then the node **p** is the root of the UID-local area containing **n** and the local index of **p**, which shows the order of the node in the upper area, can be obtained from the table $\mathscr{K}$ (steps 7-9). Otherwise, **p** is not an UID-local root node (step 10). We illustrate this algorithm through Example 7.

Algorithm 2: *r*UID computation of the parent node

Input: $\mathscr{T}$, κ and $\mathscr{K}$, and the *r*UID (g_i, l_i, r_i) of a node
Output: The 2-level *r*UID (g, l, r) of the parent node

```
1.  if ($g_i$ == 1 && $l_i$ == 1) then return $null$
2.  if ($r_i$ == $true$) then
3.      $g$ := $max(\lfloor (g_i - 2)/\kappa + 1 \rfloor$, 1)
4.  else $g$ := $g_i$ endif
5.  get the fan-out $k_j$ from the row with
        the global index $g$ in $\mathscr{K}$
6.  $l$ := $\lfloor (l_i - 2)/k_j + 1 \rfloor$
7.  if ($l$ == 1) then
8.      set $l$ equal to the local index
          of the row with the global index $g$ in $\mathscr{K}$
9.      $r$ := $true$
10. else $r$ := $false$ endif
e.  return ($g$, $l$, $r$)
```

Example 7 *Suppose that κ equals 4 and the table $\mathscr{K}$ is given in Table 4.1. Let* **n** *and* **p** *denote a node and its parent node, respectively. We consider several possibilities of the* **n***:*

- **n** *is the non-root node (2, 7, $false$): From the second line of $\mathscr{K}$, the local fan-out of the UID-local area containing* **n** *is found to be two. The local index of* **p** *is equal to $\lfloor (7-2)/2+1 \rfloor$, or 3. Hence,* **p** *is (2, 3, $false$).*

- **n** *is the root node (10, 9, true): The identifier of the upper UID-local area containing* **p** *is equal to $\lfloor (10-2)/4+1 \rfloor$, or 3. From the third line of $\mathscr{K}$, the local fan-out of the UID-local area is found to be three. The local index of* **p** *is $\lfloor (9-2)/3+1 \rfloor$, or 3, greater than one, hence* **p** *is (3, 3, $false$).*

Note that if the value κ together with the table $\mathscr{K}$ are loaded into the main memory, then Algorithm 2 can be performed completely inside the main memory without any disk I/O.

4.2.3 Description of *n*-level *r*UID

The *n*-level *r*UID is the generalized concept of the 2-level *r*UID. The idea is that the frame in the 2-level *r*UID is to be considered as an “original tree”, and a new frame of this tree will be constructed in order to establish the 3-level *r*UID, and so on. The *n*-level *r*UID is useful when the size of the frame is large. Let us refer $\mathscr{T}$ and the frames recursively built one upon the other as the data levels. We enumerate the levels such that the original $\mathscr{T}$ is the level one, its frame is the level two, and so on.

Definition 7 *(n-level rUID) Given an XML tree $\mathscr{T}$, the m-level rUID of a node* **n** *in $\mathscr{T}$ is defined as $< g, (l_{m-1}, r_{m-1}), \cdots, (l_2, r_2), (l_1, r_1) >$ where:*

- *for j = 1 $\cdots$ m-1: l_j is the local index and r_j is the root indicator of* **n** *in its UID-local area identified by $< g, (l_{m-1}, r_{m-1}), \cdots, (l_{j+1}, r_{j+1}) >$ in the level j+1.*
- *g is the original UID in the level m.*

The g, l_i, and r_i (i=1$\cdots$m-1) are similar to the first, second, and third components of 2-level rUID.

The global parameters of the *n*-level *r*UID is comprised of the global parameters of all levels.

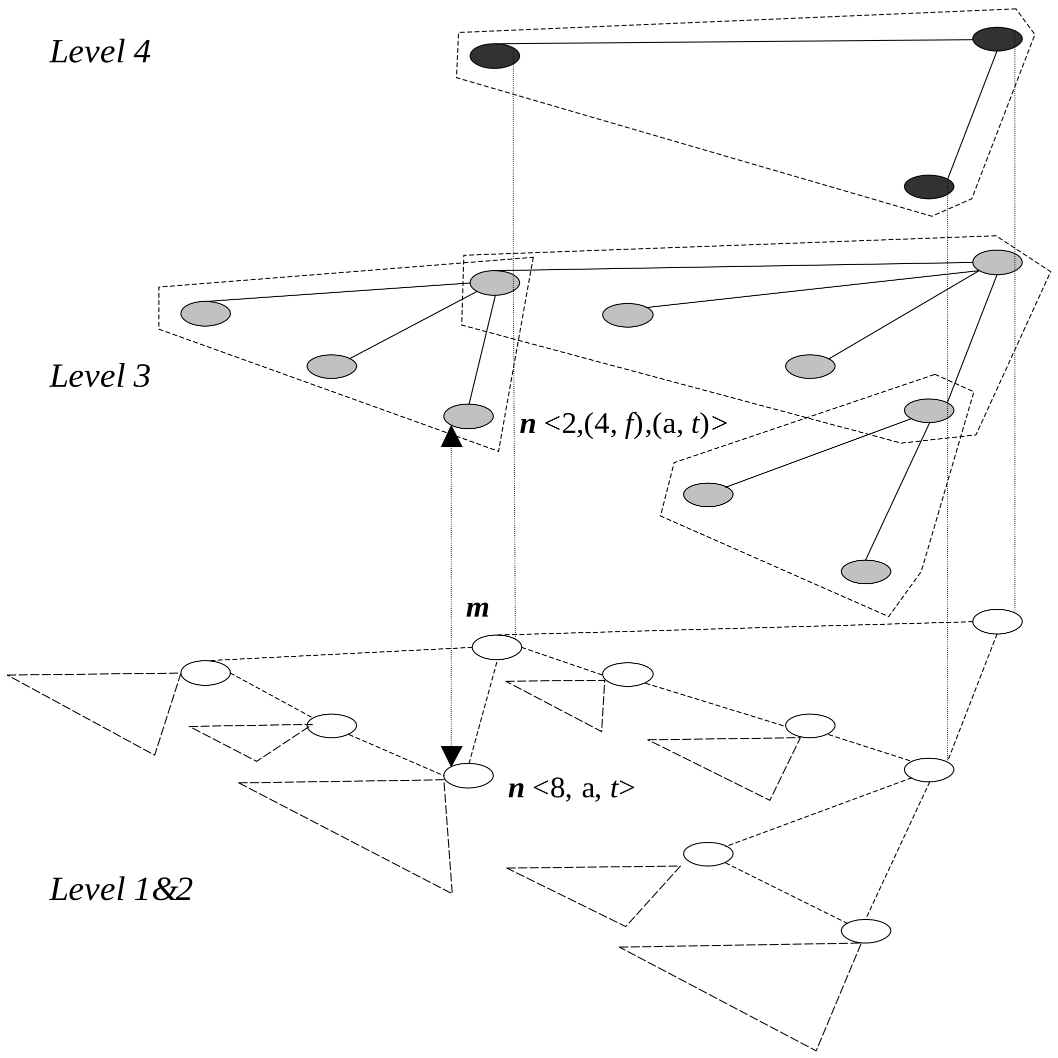


Figure 4.4. *n*-level *r*UID architecture.

Example 8 *In Figure 4.4, each polygon denotes an UID-local area. Suppose using 2-level rUID the node* **n** *has the identifier <8, (a, true)>, where the boolean value true indicates that* **n** *is the root of an UID-local area, 8 is the global index of* **n** *in the second level's frame, and the integer number a is the index of* **n** *in the UID-local area rooted at* **m** *of the same frame. In 3-level rUID, the index 8 is decomposed into (2, 4, false) and the 3-level rUID of* **n** *is <2,(4, false),(a, true)>.*

Construction of *n*-level *r*UID: For a large XML tree, the UID levels are constructed consecutively, each created on the top of the previous level. First, the 2-level *r*UID of the form $<x_1,(l_1,r_1)>$ is constructed. If needed, the 3-level *r*UID of the form $<x_2,(l_2,r_2),(l_1,r_1)>$ is constructed, and so on. The process stops when the top level becomes small enough. In practice, encoding a large XML tree requires only a few levels.

4.2.4 Properties of *n*-level *r*UID

The *n*-level *r*UID has several properties, which are crucial for a numbering scheme to be applicable to the management of a large amount of XML data.

Scalability

Theoretically, the proposed *r*UID can present the identifiers of nodes for any tree since this technique can be applied recursively. In our experiments in Section 4.4, it is sufficient to use 2-level *r*UID to enumerate the XML data sets.

Robustness in Structural Update

In *r*UID, the scope of identifier update resulted from a node insertion is reduced in comparison with UID. If a node is inserted, at first only the nodes in the UID-local area where the update occurs need to be considered. If a free space is available for the new node, then among the descendants of the sibling nodes to the right of the inserted node, only those which belong to the same UID-local area will have their identifiers modified. The nodes in the descendant areas are not affected because the frame $\mathscr{F}$ is unchanged. Otherwise, if such a space does not exist then the fan-out of the tree used in enumerating the UID-local area must be enlarged. Rather than modifying the identifiers of every XML component, the enlargement changes only the identifiers of the nodes in this area. In both cases, since the size of an UID-local area is much smaller than the size of the entire data set tree, the scope of the identifier update is greatly reduced.

Structural Relationship Determination

The *r*UID preserves an important property of the original UID whereby given the identifier of a node the parent node's identifier can be computed entirely in the main memory without any I/O. Therefore, the ancestor and parent axes of a given node can be constructed. Note that the construction is available only with the numbering schemes that enable the computation of the identifier of the parent node of a node. This property facilitates the evaluation of the structural joins in XML queries and is also useful for the fast reconstruction of a portion of an XML document from a set of elements.

4.3 SKEYRUS - a System of 2-level *r*UID

There are many possible approaches to implementing the idea of *r*UID. In this section, we present a prototype system named SKEYRUS that applies *r*UID to extend the search operation by keywords on XML data, a common request to Web applications, by integrating the hierarchical order of the data items. The features of our implementation are the simplicity and the efficiency. All modules of the systems are built in Java, including the key/data search engine using the B-tree data structure, and run in the same address space in the main memory. Therefore, no inter-process communication between processes is required.

4.3.1 System Design

The system design of SKEYRUS is depicted in Figure 4.5. SKEYRUS has been built based on Java technology and freewares. The SAX parser from Xerces software foundation [4] is used for XML data parsing. The indexes of data in B-tree structure are built using BerkeleyDB database engine [41] that is an embedded database library linked directly into the application. Since the library runs in the same address space, no inter-process communication is required for database operations. Berkeley DB uses key/data pairs to identify elements in the database. Berkeley DB supports hash tables, B-trees, simple record-number-based storage, and persistent queues. Key and data items can be arbitrary binary data of practically any length. There is a single data item for each key item, by default, but databases can be configured to support multiple data items for each key item. In SKEYRUS, we used the B-tree indexing structure and set the mode of multiple data items for each key item.

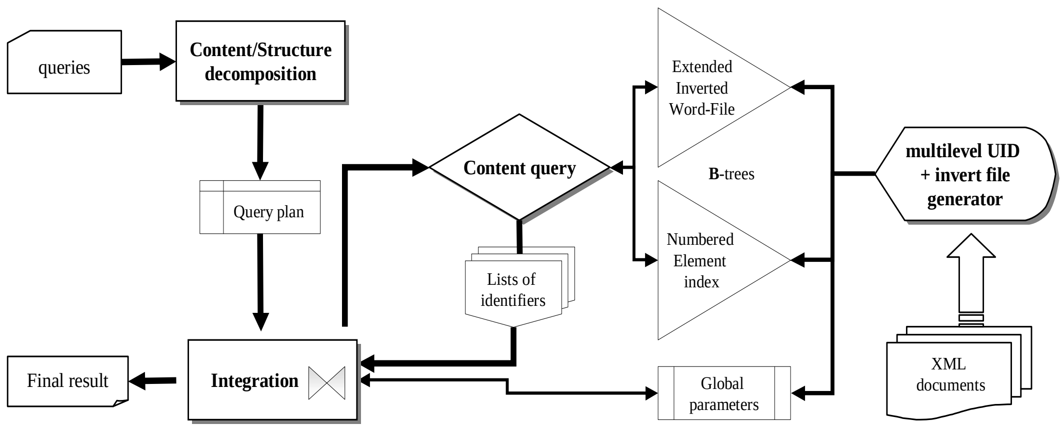


Figure 4.5. System design of SKEYRUS.

The input queries of SKEYRUS are expressed by XPath [48] expressions having word-containment predicates. XPath is a language that provides basic facilities for manipulation of strings, numbers and boolean in the logical structure of an XML document. One important kind of XPath expression is *location path*, which selects a set of nodes relative to a context node. A location path has a number of location steps, where an initial node-set is generated from an axis and a node test, and then filtered by each of the predicates in turn. The input queries of SKEYRUS are those expressions that contain axes specifying the node position in XML documents and the predicate of word-containment (i.e the *contains*() function). An example of the queries for SKEYRUS is:

```
/child::doc/child::chapter/descendant
::figure[contains(child::title,‘genetic’)]
```
(2)

which selects the figure, the title of which contains the keyword ‘genetic’, of a chapter of a document.

Processing the structural part of the queries is performed based on 2-level *r*UID. Evaluating a location step of a query involves determination of the candidate sets, which contain either elements or attributes, using a data structure that supports keyword search, and joining the sets via the axis of the location step.

An initial *query plan* is generated by parsing the input into location steps described by the attributes: the step identifier **ID**, the candidate node sets to be joined **ns-I** and **ns-II**, the axis of the joining **axis**, the function to filter the second candidate node set **qf**, and an output indicator (**O**). A query plan of the XPath expression (2) is shown in Table 4.2.

ID	**ns-I**	**axis**	**ns-II**	**qf**	**O**
1	root	child	doc		
2	doc	child	chapter		
3	chapter	descendant	figure		yes
4	figure	child	title	contains ‘genetic’	

Table 4.2. A query plan for the query (2).

4.3.2 Data Structure in Secondary Memory

We used three tables named globalPara, codeName, and codeWord in SKEYRUS. The input XML documents are parsed and the tree nodes are numbered using *r*UID. The global parameters κ and $\mathscr{K}$ generated in the process are stored in the table globalPara. For each element and attribute, a tuple:

```
<rUID, EleAttrName, descr>
```

is generated and saved in the tables codeName, where `rUID` field consists of the global index, local index, and root indicator. The `rUID` is the identifier of the element or attribute `EleAttrName`, and `descr` contains the node type of `EleAttrName` and its position, which is the address where data resides, in XML documents.

For each leaf node, an inverted file structure for the words in the node content is created and consists of a number of tuples:

```
<rUID, word, descr>
```

stored in the tables codeWord. The `rUID` is the identifier of the element or attribute containing the word `word`. In SKEYRUS, since the hierarchical level can be computed using *r*UID, we can discard the *depth* parameter that has been introduced in [13]. For a given word, the inverted file structure allows to list the identifiers of all the elements or attributes containing the word.

For both element and word, the data is indexed such as given an element name or a word, the list of the *r*UID of elements corresponding can be found fast. The data is indexed by B-tree access method that supports the creation of multiple data items for a single key item (i.e. element name or word). The duplicate records are maintained in sorted order not only by the key field but also by the data field. The Berkeley DB engine can be configured to sort duplicate data items by setting the corresponding flags like:

```
table.set_flags(Db.DB_DUP);

table.set_flags(Db.DB_DUPSORT);
```

In such manner, for a given element name or word, the related tuples are maintained sorted by *r*UID.

4.3.3 Main Structural Joining Mechanism

This module joins, using *r*UID as the key, the tables in main memory according to given axes. The *join mechanism* consists of three main steps as follows:

Axis construction: The nodes that belong to a specific axis of any node in a given set are determined by their identifier. The axis construction is performed in main memory without any disk I/O.

Sorting: The nodes of a given list are sorted based on the *r*UID components.

Matching: The pairs of identical nodes from two lists are extracted. If the lists are sorted then only an one-scanning for each list is needed to select the pairs from the lists.

Let us explain how these steps are used in processing a query by an example. Let A and B denote lists of elements of the names **a** and **b**. To process the query "`a/child::b`", A and B are loaded from the second memory. We maintain the data in DB files sorted, hence A and B are sorted by *r*UID. Then, rather than checking all possible pairs from tables A and B, we generate parentAxis(B) and store the finding in an buffer C. Next, we sort C by *r*UID and then match A with C. The mechanism is depicted in Figure 4.6(a), where $\mathbf{a}_i$ is the last element that matches $\mathbf{b}_j$ in the considered axis.

This join mechanism can be applied to both ancestor and parent axes. Intuitively, if a query contains the predicates on keyword then these predicates will be evaluated first to reduce the cardinalities of A and B.

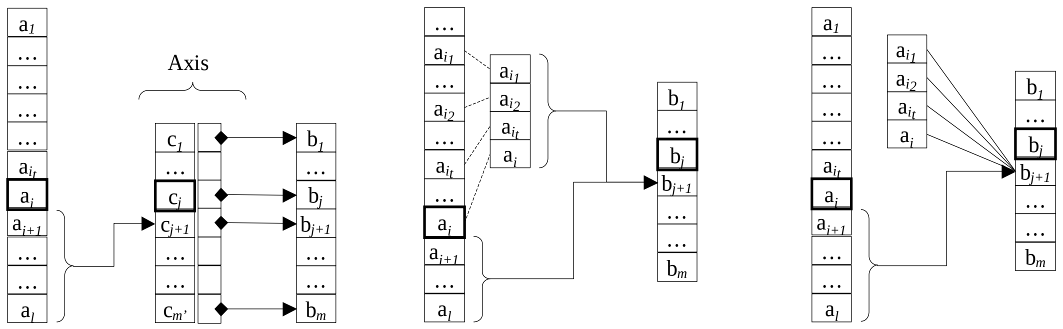
The technique can produce the qualified elements either from A and B. Normally, the matching A with C can output the qualified elements from A. With auxiliary pointers added to the buffer containing parentAxis(B) that point back to B, the output is the qualified elements from B, instead of from A. In both cases, the number of buffer arrays is at most four. For example, the list A is stored in the buffer 1, B in the buffer 2, parentAxis(B) before and after sorting in the buffer 3. The output from the buffer 2 produced by the matching buffers 1 and 3 is stored in the buffer 4.

4.3.4 Auxiliary Join Mechanisms

The ability to construct the parent and ancestor axes supports the main join mechanism discussed in Section 4.3.3. In this part, we present other join mechanisms used in SKEYRUS that do not require the axis construction ability of *r*UID.

In a well-formed XML document, a node is an ancestor of another node if the region, bounded by the start tags and the end tags, of the former covers the region of the latter. The ancestors of a node consist of its parent and its parent's parent and so on. A node is a preceding node of another node if the end tag of the former is located before the start tag of the latter.

Given two nodes **p** and **q**, let us denote $\mathbf{p} \prec_a \mathbf{q}$ iff **p** is an ancestor of **q**, $\mathbf{p} \prec_{a+} \mathbf{q}$ iff



(a) Axis construction with backward pointer

(b) Ancestor scan with current ancestor buffering

(c) Preceding scan with sorting by $<_{preorder}$

Figure 4.6. Join mechanisms used in SKEYRUS.

$\mathbf{p}$ is an ancestor of $\mathbf{q}$ or $\mathbf{q}$ itself, $\mathbf{p} \prec_p \mathbf{q}$ iff $\mathbf{p}$ is a preceding node of $\mathbf{q}$, $\mathbf{p} \prec_{preorder} \mathbf{q}$ iff $\mathbf{p}$ is an ancestor or preceding node of $\mathbf{q}$.

Note that the relationship $\prec_{preorder}$ is a total order. The following lemmas can be used to improve the efficiency of $\prec_a$ and $\prec_p$ joins for two lists of nodes sorted by $\prec_{preorder}$:

Lemma 1 *For two nodes* $\mathbf{n}_1$ *and* $\mathbf{n}_2$ *of an XML tree such that* $\mathbf{n}_1 \prec_{preorder} \mathbf{n}_2$, *if* $\mathbf{a} \prec_a \mathbf{n}_2$ *then either* $\mathbf{a} \prec_{a+} \mathbf{n}_1$ *or* $\mathbf{n}_1 \prec_{preorder} \mathbf{a}$.

Proof: Because $\mathbf{n}_1 \prec_{preorder} \mathbf{n}_2$, either $\mathbf{n}_1 \prec_a \mathbf{n}_2$ or $\mathbf{n}_1 \prec_p \mathbf{n}_2$. In the first case, the ancestor nodes of $\mathbf{n}_2$ include the ancestor nodes of $\mathbf{n}_1$, the node $\mathbf{n}_1$ itself, and the nodes in the path connecting $\mathbf{n}_1$ and $\mathbf{n}_2$, exclusively. In the second case, let $\mathbf{l}$ is the lowest common ancestor of $\mathbf{n}_1$ and $\mathbf{n}_2$. The ancestor nodes of $\mathbf{n}_2$ include nodes in the path connecting the root node and $\mathbf{l}$, the node $\mathbf{l}$, and the nodes in the path connecting $\mathbf{l}$ and $\mathbf{n}_2$, exclusively. Because in a tree, there is unique path connecting two nodes, the nodes exclusively belonging to the path connecting $\mathbf{l}$ and $\mathbf{n}_2$ also are following nodes of $\mathbf{n}_1$. In both cases, the Lemma holds. □

From Lemma 1, if lists A and B are sorted by $\prec_{preorder}$ and $\mathbf{a}_i$ is the last node in A such that $\mathbf{a}_i \prec_{preorder} \mathbf{b}_j$ then for looking for the ancestors of the node $\mathbf{b}_{j+1}$, we have to look at the ancestor of $\mathbf{b}_j$ (i.e. $\mathbf{a}_{i_1},\cdots,\mathbf{a}_{i_t},\mathbf{a}_i$), and the nodes after $\mathbf{a}_i$ in A (i.e. $\mathbf{a}_{i+1},\cdots,$ $\mathbf{a}_l$), Figure 4.6(b).

Given two nodes $\mathbf{n}_1$ and $\mathbf{n}_2$ of an XML tree such that $\mathbf{n}_1 \prec_{preorder} \mathbf{n}_2$, if $\mathbf{n} \prec_{preorder}$ $\mathbf{n}_1$ then $\mathbf{n} \prec_{preorder} \mathbf{n}_2$. Therefore, if lists A and B are sorted by $\prec_{preorder}$, we need to scan once the lists to select the pairs $\{\mathbf{a}, \mathbf{b}\}$ from the lists such that $\mathbf{a} \prec_{preorder} \mathbf{b}$. In Figure 4.6(c), the preceding nodes of $\mathbf{b}_j$ (i.e. $\mathbf{a}_{i_1},\cdots,\mathbf{a}_{i_t},\mathbf{a}_i$) are also the preceding nodes of $\mathbf{b}_{j+1}$.

4.3.5 Content Processing

In SKEYRUS, the content processing module loads the identifiers of elements or attributes having a given name string and containing a given keyword. Specifically, from the tables codeName and codeWord, the rUIDs of the tuples where EleAttrName = '*nameString*' and word = '*keyword*' are retrieved. The output of the content processing module is stored in buffer tables in main memory.

Given an element list A and a keyword list W, the matching of A and W on *r*UID produces the list of elements that contain a word from W. The matching is used to process the predicate on keywords in the input query for SKEYRUS.

4.3.6 Database Table Decomposition

When the data is large, we have to distribute the tuples into smaller tables to speed up query processing, each table has a name that reflects its content. Specifically, the table codeName can be decomposed into the tables, the elements in each of them have the same tag name that also is the name of the table itself. Since the number of word string values in an XML data set may be much larger than the number of tag names of the elements and attributes in the data set, in order to keep the number of tables manageable, the table codeWord can be decomposed by a *common prefix* of words rather than by the word values themselves, where the common prefix means a shared prefix of words. For example, using 2-character common prefix decomposition, the tuples of the table codeWord that correspond to the words 'computer' and 'control' are stored in the table of the codeWord type that has the name "CO". In this chapter, we run the experiments in 4.4.2 (4.4.2) without (with) table decomposition.

4.3.7 Frame selection

In our experiments, we use Algorithm 3 to select the frame for an XML tree. The high threshold guarantees the expression of the local index in an UID-local area by an integer value. The low threshold is used to avoid unnecessary creation of small areas whereas these areas can be grouped together. The grouping accelerates the determination of ancestor-descendant relationship.

Algorithm 3: Frame selection

Input: An XML tree $\mathscr{T}$, a high threshold h, a low threshold o,
Output: A frame of $\mathscr{T}$

```
1. cNode := root node
2. while the root node is not marked
3.   compute the size of area rooted at cNode
     and the number of the nodes in the area
4.   if the size <= h && the number => o
5.     mark cNode
6.   else
7.     run the procedure with child nodes of cNode
     endif
9.   if (not found any area)
10.    reduce the low threshold
11.  else reset default the low threshold endif
   endwhile
```

In the step 3 of Algorithm 3, by the size of an area we mean the largest UID possibly needed to enumerate the nodes of the area.

4.4 Experiment

In this section, we describe several performance experiments with SKEYRUS to evaluate the scalability and the effectiveness of *r*UID in XML query processing. For these purposes, we generated three synthetic datasets. The specification of the synthetic datasets is shown in Table 4.3. The first two datasets supposedly describe the personnel organization of a company using the DTD called “personnel.dtd” described in Appendix C. The dataset *I* has a small size but an unbalanced structure and a high degree of recursion. In addition, we generated the third data sets using XMark [2], the core DTD of which is shown in Appendix D, Figure 6.3. The second and third datasets are used as the main dataset for query processing test. We believe that an appropriate decomposition of XML data, the investigation of which is out of the scope of the research, will save the unnecessary comparisons for structural joins. An example of such comparison is the check for the hierarchical orders of two elements belonging to two unrelated data parts. Therefore, the size and the number of element and attributes of the second and third datasets is regarded to be sufficient for our experiments.

Data set	**Size**	**# elem&attr**	**DTD**
I	7.6Kb	201	personnel.dtd
II	3172Kb	50052	personnel.dtd
III	23.4Mb	413111	auction.dtd

Table 4.3. Specification of the data sets.

The experiments in our study were conducted on a workstation running on Windows XP Professional with 2GHz CPU, 2GB of RAM and a hard disk to store data.

4.4.1 Scalability and Robustness of *r*UID

The original UID technique failed to numerate entirely the data sets *I* and *II*, e.g. the identifier value exceeded the maximal value of integer supported by programing languages. For example, Java provides the 8-byte integer type ranging from $-9223372036854775808L$ to $9223372036854775807L$. While can deal with Shakespeare’s plays[1], the UID technique was overflowed when the plays were grouped in a *collection*, for example by adding a DTD declaration $<$`!ELEMENT collection (play+)`$>$. Meanwhile, the 2-level *r*UID succeeded in enumeration of all of these

[1]Available at http://sunsite.unc.edu/pub/sun-info/xml/eg/shakespeare.1.10.xml.zip

cases since 2-level *r*UID can use the nearly double number of bytes to present the identifiers of nodes in an XML document.

In our method, it is desirable to use maximally the possible value of local index to reduce the number of UID-local areas. To represent the *r*UIDs of the data set *III*, the maximal global and local indices are equal to 27843 and 16379869450275314L, respectively.

In addition, as discussed in Section 4.2.4, the design by levels of *r*UID reduces the scope of the identifier change due to a structural update. For example, when a new employee **`person`** is inserted, it is likely that only the identifiers of the nodes in the UID-local area containing the manager **`person`**, who manages the new employee, are altered.

4.4.2 *r*UID and XML Query Processing

We compared query processing of SKEYRUS with Xalan [4] and the method in [13], denoted by IKS[13].

Comparison with Xalan

In this experiment, the tables codeName and codeWord have not been decomposed. On the data set *II*, we conducted the experiments using the input XPath expressions that are shown in the abbreviated syntax in Table 4.4 of the following categories:

simple: XPath expressions having one or two parent and ancestor axes without any keyword, e.g. P_1 and P_2.

wildcard XPath expressions having parent and ancestor axes and require the hierarchical level, e.g. P_3 and P_4.

complex XPath expressions having both structural search and keyword search, e.g. P_5 and P_6.

The performance test on the data set *II* with the query set A is shown in Figure 4.7, where S_q denotes the time needed for processing the structural joins in the main memory of SKEYRUS, *Skeyrus* denotes the total elapsed time (i.e. including the time

Queries	**XPath expressions**
P_1	`//person`
P_2	`//person/email`
P_3	`//person/*/person`
P_4	`//person/*/*/person`
P_5	`//person[contains(note,’Academic’)]`
P_6	`//person/name[contains(given,’Michael’)]`

Table 4.4. Query set A for the data set *II*

for loading data) of SKEYRUS, X_q denotes the time needed for processing the structural joins in the main memory of Xalan, and $Xalan$ denotes the total elapsed time (i.e. including the time for loading data) of Xalan.

XMark benchmark is provided with a set of test queries. For the purpose of the research, we consider only the structural part of these queries as shown in Table 4.5 as follows:

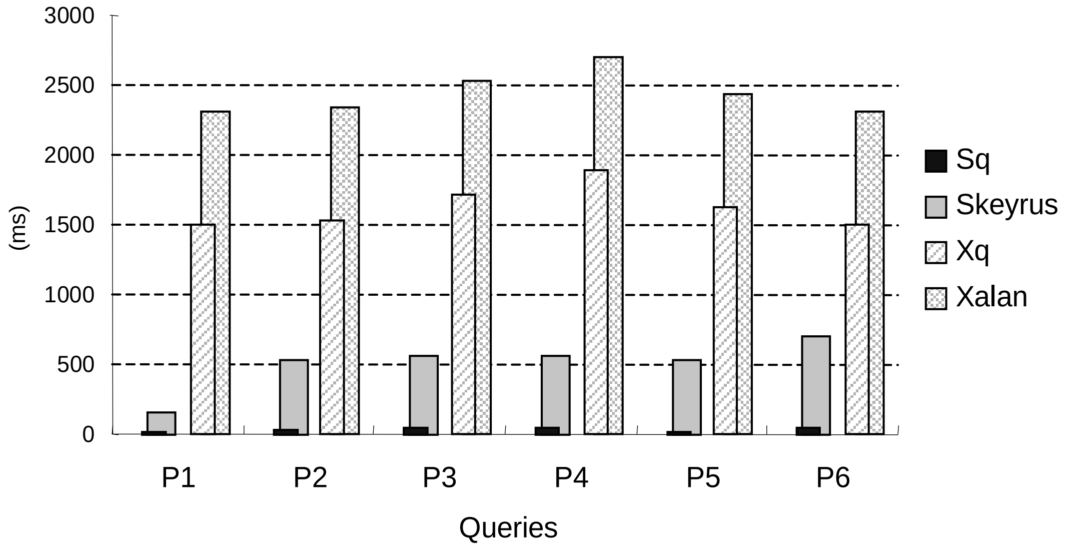


Figure 4.7. Elapsed times for the queries P_{1}-$_{6}$ on dataset II.

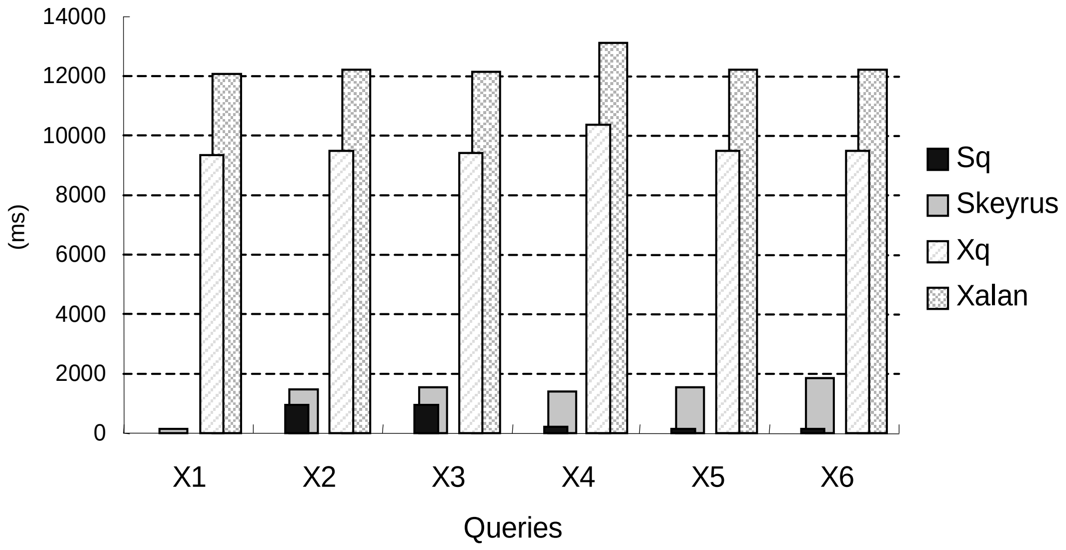


Figure 4.8. Elapsed times for the queries X_{1}-$_{6}$ on the dataset III.

Queries	XPath expressions
X_1	`/site/regions`
X_2	`/site/regions/europe/item`
X_3	`/site/regions/australia/item`
X_4	`//open_auction//description`
X_5	`/site/people/person`
X_6	`/site/people/person/profile`
X_7	`/site/people/person/profile/interest`
X_8	`/site/open_auctions/open_auction`
X_9	`/site/open_auctions/open_auction/initial`
X_{10}	`/site/closed_auctions/closed_auction`
X_{11}	`/site/closed_auctions/closed_auction/annotation/description /parlist/listitem/parlist/listitem/text/emph/keyword`
X_{12}	`//annotation/description/parlist/listitem/parlist/ listitem/text/emph/keyword/`

Table 4.5. Query set B for the data set *III*

These queries ranges from simple to extremely long queries, including both parent and ancestor axes. The performance test on the data set *III* of the query set B is shown in Figs. 4.8 and 4.9.

From the experiments on both datasets *II* and *III* using “personnel.dtd” and “auction.dtd” DTDs, it is clear that SKEYRUS is effective in processing the queries. For most of queries evaluated, the processing time required by SKEYRUS is 20-30% of the time required by Xalan. For the query with a large number of structural joins, for example X_{11} and X_{12}, query processing of SKEYRUS remains satisfied.

In general, the time for loading the data used in processing a query of SKEYRUS is small in comparison with XALAN because SKEYRUS loads only the portion of data needed for the query. It does not require that the whole data must reside in the main memory.

SKEYRUS deals well with the queries where the names of several elements participating in the axes of the queries are unknown by using the ability of *r*UID for hierarchical level determination. The required time for processing such queries is not increased sharply since considering all of possibilities of the elements is not necessary.

The main contribution to the effectiveness of the SKEYRUS is the ability of *r*UID to perform the structural joins efficiently. As shown in Figs. 4.7, 4.8, and 4.9, the time for processing in the main memory, S_q, are small parts of the total elapsed times.

Comparison with IKS[13]

To compare SKEYRUS with the method proposed in IKS[13], we use the queries P_5 and P_6, that have both structural search and keyword search, in Query set A for the the data set II and the queries X_{13} and X_{14} in Query set B for the data set III. These queries are listed in Table4.6. In this experiment, the data have been decomposed using the element and attribute tag names and the 1-character prefix of words, as discussed in Section 4.3.6.

Queries	**XPath expressions**
P_5	`//person[contains(note,'Academic')]`
P_6	`//person/name[contains(given,'Michael')]`
X_{13}	`//description[contains(text,'house')]`
X_{14}	`//categories//listitem[contains(text,'discover')]`

Table 4.6. Query set for the comparison with IKS[13]

According to the method described in [13], for each tag name that appears in the XML data, a binary table of the structure `tag_name`(`tag_nameID`, `child_elementID`, `value`) is used to represent the relationship of the elements having the tag name and their child elements. The collection of these binary tables represent the structure of an XML document. In addition, the inverted files `word_type`(`elementID`, `depth`, `location`) are used to express the occurrence of words in XML elements. All these tables are stored in Oracle 9*i* Personal Edition for Windows.

The total inverted file can be decomposed into smaller files clustering by the elements (or attribute) name and the words (a prefix or the entire) that appear in the elements (or attributes). The decomposition can speed up keyword searching. However, the method can produce a large number, thousands or tens thousands, of tables and the number of table may be a concern. In addition, to deal with the keyword search, where the containing element (or attribute) is not specified explicitly, e.g. "`//*[contain...]`", the tables corresponding to the word have to be concatenated before further processing. In this section, SKEYRUS and IKS[13] are treated equally, without the above decomposition.

Note that the approach proposed by IKS[13] to store the elements of an XML document in binary tables is not efficient to process the queries with the ancestor-descendant relationship, i.e. '//'. For example, to process a query like `//categories//listitem`, we have to join the tables that correspond to the edges along all possible paths connecting **`categories`** and **`listitem`**. The number of such paths can be large and it is hard to find all of them. Therefore, we cannot include the performance test of the [13] method for the query X_{14} in our experiment.

The performance test for the queries P_5, P_6 in the data set *II*, and X_{13} in the data set *III* is shown in Figure 4.10, where *Skeyrus* and *IKS* denote the times needed for processing and the total elapsed time (i.e. including the time for loading data, the intercommunication among processes) of SKEYRUS and IKS[13], respectively.

4.5 Summary of Chapter 4

Queries on keywords are common user requests whereas XML makes the queries on document structure available. Therefore, it is natural to integrate these tasks to enrich the search selectivity of users. In this study, our approach to achieve the goal is application of a numbering scheme. We proposed the novel recursive numbering scheme *r*UID that is robust in structural update, scalable for coding arbitrarily large XML documents, and expressive in presenting the main XPath axes. Note that *r*UID is robust in the terms of structural update without *clue*, i.e. without the DTD or XML Schema. In the next chapter, we will discuss another technique that takes into account of the schematic information from DTD or XML schema.

The proposed *r*UID has been implemented in SKEYRUS, a system enables the searches on both keyword and structure on XML data. The performance experimental results have shown the effectiveness of *r*UID.

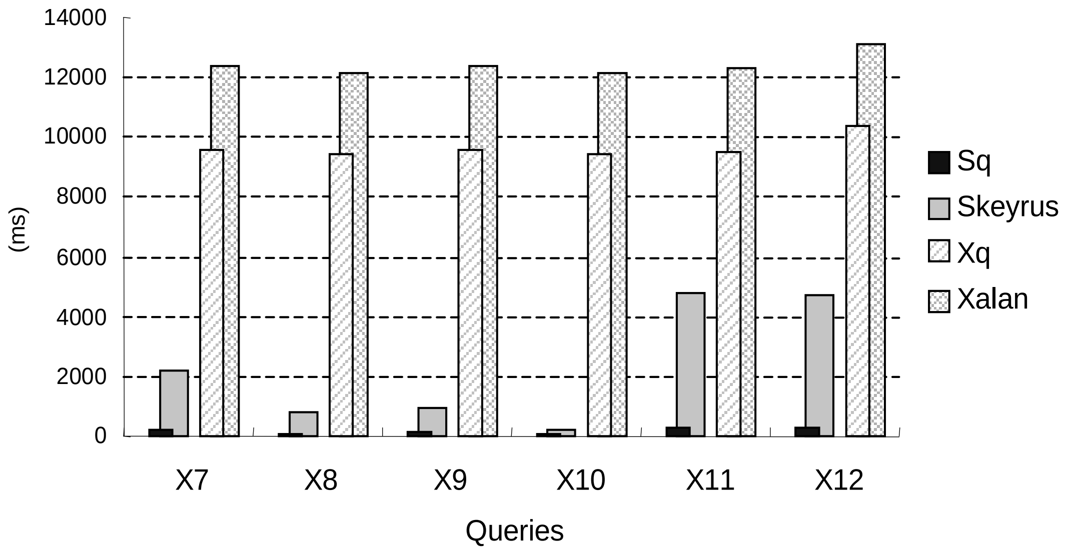


Figure 4.9. Elapsed times for the queries X_{7-12} on the dataset III.

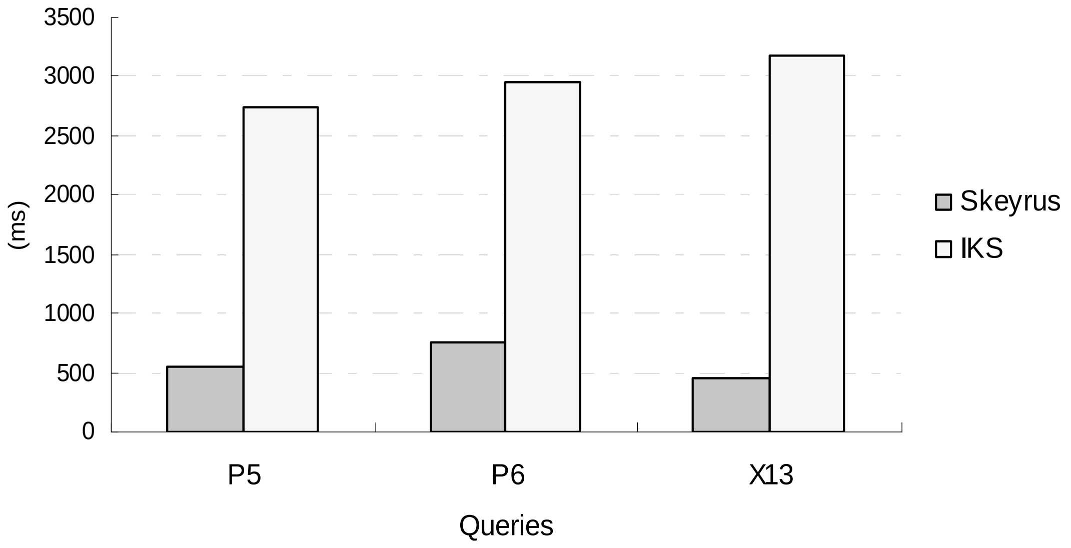


Figure 4.10. Elapsed times for the queries P_5, P_6, and X_{13}.

Chapter 5

XML Query Processing using Virtual Joins

5.1 Introduction

In this chapter, we investigate the role of DTD and XML Schema in XML query processing. Originally, DTD and XML schema have been designed to serve as a mean to validate XML documents. However, we found that the structural information extracted from the document definitions can be used to improve the efficiency of XML query processing.

XML [47] data has a recursive tree structure, which can be represented by a rooted label tree. The structure of XML documents in a class can be described by a grammar that is a set of rules of the hierarchical relationship among XML elements in the instance documents of this class. The hierarchical order of elements in an XML document physically depends on their location in the physical storage structure of the document. On the other hand, the elements have to obey the *schematic* order described in the grammar associated with the class containing the document.

Queries on XML data typically specify elements by selection predicates and their tree structure relationship. There are a number of proposed methods for verifying the structure relationship of XML elements. An enumeration that allows the identifier of the parent element to be computed from the identifier of a child element has been presented in [52], hence the "`parent-child`" relationship can be determined using a calculation on the identifiers. The oversize length of identifier and the lack of ability to determine the tag name of the parent element limit the efficiency of this numbering scheme in query processing. The pattern of node paths has been used in [26] to select the elements having the similar hierarchical order. In [40, 31, 35, 27], the 3-tuple (`startPos`, `endPos`, `level`) and equivalent tuples have been used to present the hierarchical order of XML elements in an XML document. The recent approaches to the problem use the structural joins, which select the pairs of XML elements from the candidate sets such that a given hierarchical order holds.

Note that most of XML documents in use are associated with a DTD or XML schema. Since the descriptions integrate the document structure with data types and define the relation of schemata to XML document instances, the XML documents exchanged over the web can share their grammars efficiently. However, the prior techniques to process XML queries have not utilized the information about the schematic structure of XML documents expressed in the descriptions. For example, the enumeration in [52] has been designed for a general tree without any restriction on its structure. Similarly, the presentation (`startPos`, `endPos`, `level`) in [40, 31, 35, 27] is extracted from an XML instance document with the assumption that the XML tree can be in any shape.

In structural joins, the indexing data of the candidate sets, a portion of which is only used to produce the partial answers, has to be provided before joining, normally by I/O access to the secondary memory. However, the previous studies have not sufficiently investigated the I/O workload needed to get the candidate sets. This study aims at the improvement of the I/O complexity for XML query processing. The issue is important since the I/O speed is much slower than the computation speed in the main memory. We observe that the schematic structure of XML documents can be used to verify the structural requirement of the elements in an XML query, without the knowledge of their actual position in the physical storage structure. Based on this observation, we propose a new approach to interpret XML queries. For example, to process the XML query "`a/b`", the prior joining techniques require the indexing data of the candidate sets $\{\mathbf{a}\}$ and $\{\mathbf{b}\}$ to be loaded then join these sets to find the pairs $\{\mathbf{a}_i, \mathbf{b}_j\}$ such that $\mathbf{a}_i$ is the parent element of $\mathbf{b}_j$. Actually, the elements $\{\mathbf{a}_i\}$ are not of the interest in the final answer of the query, which consists only of the elements $\mathbf{b}_j$. The indexing data of $\{\mathbf{a}\}$ is used just for checking the parent-child relationship of $\mathbf{a}_i$ and $\mathbf{b}_j$. Using the new approach, rather than joining the elements sets $\{\mathbf{a}\}$ and $\{\mathbf{b}\}$, for each $\mathbf{b}_j$ we seek a method to check if there exists $\mathbf{a}_i$ that is the parent element of $\mathbf{b}_j$ using the indexing data of $\mathbf{b}_j$ only. Therefore, we can save the I/O workload for loading the indexing data of the set $\{\mathbf{a}\}$. Figure 5.1 illustrates the difference of the prior approaches and our approach in processing the XQuery expression:

```
FOR $b IN /site/people/person/[address='Japan']
RETURN $b/city/text()                                        (S1)
```

The prior approaches require the indexing data of six elements **`site`**, **`people`**, **`person`**, **`address`**, **`name`**, and **`text`**. Using the new approach, the indexing data of only two elements **`address`** and **`text`** is required.

5.1.1 Contribution

This study aims at the improvement of the I/O complexity for XML query processing. The issue is important since the I/O speed is much slower than the computation clock in the main memory. We propose a mechanism called Virtual Join for XML query processing that utilizes the information about the schematic structure extracted from the DTD or XML schema of XML documents. The core of the mechanism is a Structure Coding for XML data, SCX, that compactly expresses the schematic structure of XML elements. The SCX has the following property:

> If the tag name and the structural code of an element are known, the tag name and the structural code of the parent element can be determined without any I/O.

Therefore, for a given element, the tag names of all of its ancestor elements can be determined recursively. Note that the tag names of elements are essential for evaluating XML path expressions. Our study provides evidence that the information about the schematic structure of XML documents declared in DTD or XML schema can be used effectively in the indexes for XML data. In addition, it shows that different indexing techniques can be integrated to complement each other to improve the XML query processing.

We present the preliminaries of our study in Section 5.2. Our main contribution is presented the next three sections. In Section 5.3, we give the definition and the construction algorithm for SCX that enumerates XML elements based on the schematic structure deduced from DTD or XML schema. Incorporating both structural and tag

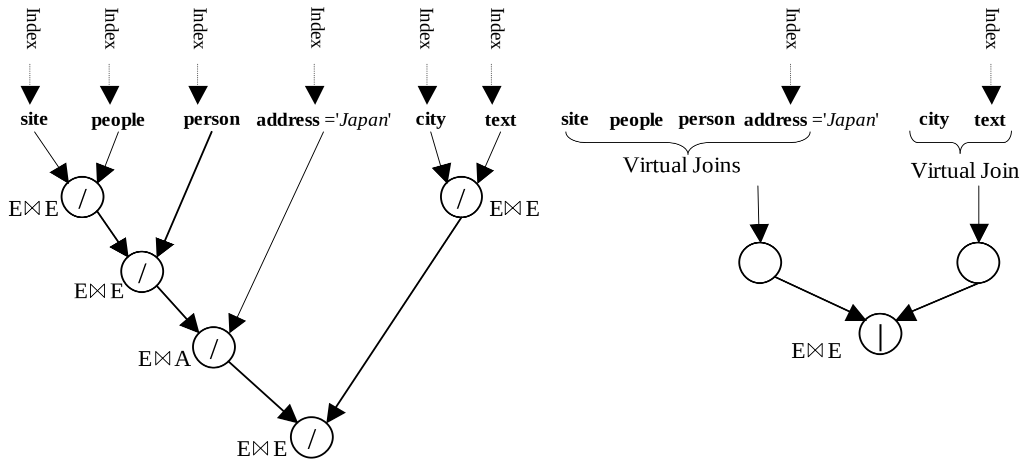


Figure 5.1. The prior and new approaches for processing the query S_1

name information, SCX allows the navigation to the parent of an element, as well as testing the tag name of the parent in the time independent of the document size. In Section 5.4, we describe the Virtual Join mechanism to evaluate XML queries in both path and twig patterns. The mechanism has an optimal I/O workload: The indexing data of only the candidate sets that contain the output elements or relate with the selection predicates is needed. It does not require the indexing data to be sorted. Many intermediate joins can be avoided using the operations on SCX. In Section 5.5, we present the experimental results to show the efficiency of our method in XML query processing with various configurations. Not any special indexing structure except the B^+-tree is required. We conclude this chapter with a suggestion for the future work in Section 5.6.

The current result of this research can be primarily applied in the applications categorized into the *Native XML Databases* and *Content Management Systems* groups, as discussed in Section 2.3.1, page 23 of this thesis.

5.1.2 Related work

Querying on the structure is an essential task of the databases of semi-structure and XML data. Several structural summaries have been presented as a graph for semistructured data. Structural information, such as node paths, is extracted from the data source, classified, and then represented in a structure graph. The graph can be used both as an indexing structure and a guide by which users can perform meaningful and valid queries. A data structure called Data Guide was proposed in [32] that records all possible paths in the source semi-structure data. Paths also can be summarized grouping by type as in T-index [38]. In order to reduce the complexity of construction of structural summary, the bi-simulation mapping has been chosen instead of Data Guide as in [29]. Another approach is Index Fabric [5] that encodes paths in the semi-structured data as strings and inserts the codes in a balanced B^+-tree-*like* index that can be combined with a storage manager.

A path-based approach to query the XML data by storing all available node paths in a table of RDBMS and making queries over the pattern of the node paths has been proposed [26]. An integration of XML node numbers in query statements and an algorithm for transformation from XPath to SQL have been discussed in [21]. A general approach to store XML data in tables of RDBMS, where a query is evaluated by joining the tables containing the data items related to the query, has been presented in [24].

The presentation of XML elements by (`docID, startPos, endPos, nodeLevel`) and the equivalent tuples using `pre` and `post` orders, `preorder` and `range` have been used in [8, 31, 35, 27, 36] for processing structural joins. In our work, to present the actual order of elements in an XML document, we adopted the presentation. Both [35] and [27] use stacks in the structural join algorithms to reduce the number of match tests between candidate sets of a join. The algorithms in [35] can produce the result sorted either by ancestor or descendant nodes. The algorithms in [27] can process both of the path and twig queries, where the partial and total answers have been in compact stacks to avoid the large intermediate answers. The indexing structures such as B^+-tree and R-tree built-in in RDBMS*s* have been exploited in [40] to index the presentation values.

The current works related to our approach are [42, 19]. XML documents are embedded in a binary tree in [42], hence the depth of the binary tree is high in practice. [19] has proposed the XR-tree to manage the stab lists used to find the qualified pairs of elements in the ancestor-descendant structural join. The index permits skipping over the portions of the candidate sets that are guaranteed not to produce any match. The index implementation requires a new data structure other than the widely used B^+-tree and does not support well the parent-child relationship.

Our research investigates the whole query processing procedure, including the I/O workload for the indexing data. The function (5.4) of SCX was inspired by the UID method presented in [52], which enumerates the nodes of a tree by sequent integers, starting from one at the root. Besides the Virtual Join mechanism, the design of SCX solves the issues of coding size and robustness in structural update that limit the efficiency of the original UID method in query processing.

5.2 Structure information in DTD and XML schema

An overview of DTD and XML schema was presented in Section 2.1.2. In this part, let us consider the DTD and XML schema of an XML document in details. An XML document may have a reference to a DTD or an XML Schema, which contain the description of the hierarchical relationship of XML elements. By definition, DTD is a grammar for a class of documents. A DTD that defines the elements of the XML document graphically represented in Figure 5.2 may have the element type declarations shown in Listing XD.2.

Listing XD.2: Declarations containing schematic information in a DTD

```
<!ELEMENT personnel (company, business, person+)>
<!ELEMENT person (name, email?, person*)>
<!ELEMENT name (family, given)>
```

The first element type declaration indicates that the **`personnel`** element has one element **`company`**, one element **`business`**, and one or many elements **`person`**. This is an extension of the structure of the XML document used in Chapter 3 by integrating the information of all people together.

The hierarchical order of elements can be found also in the complex element descriptions in the XML schema of the same XML document. For example, the declaration in Listing XS.2 indicates that the complex element **`name`** has one element **`family`** and one element **`given`**.

Listing XS.2: Declarations in an XML schema

```
<xs:element name="name">
  <xs:complexType>
    <xs:sequence>
    <xs:element name="family" type="xs:string"/>
    <xs:element name="given" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The element hierarchy extracted from the DTD and XML schema for the XML document in Figure 5.2 is depicted in Figure 5.3.

In practice, the size of a DTD or XML schema is much smaller than the size of the XML documents associating with them. Normally, the sizes of DTD and XML schema are of a few KB whereas the size of XML documents are measured by MB. Let us call the number of parent-child pairs in the DTD or XML schema referenced by an XML document the *structural cardinality* of the document. The specifications of the DTDs of the XMark[1] and Shakespeare[2] data sets are shown in Table 5.1.

Data set	**DTD**	**Size**	**Stru.Card.**
XMark	auction.dtd	4304 bytes	117
Shakespeare	play.dtd	1184 bytes	44

Table 5.1. The specification of some DTDs

[1]http://monetdb.cwi.nl/xml/downloads.html

[2]http://sunsite.unc.edu/pub/sun-info/xml/eg/shakespeare.1.10.xml.zip

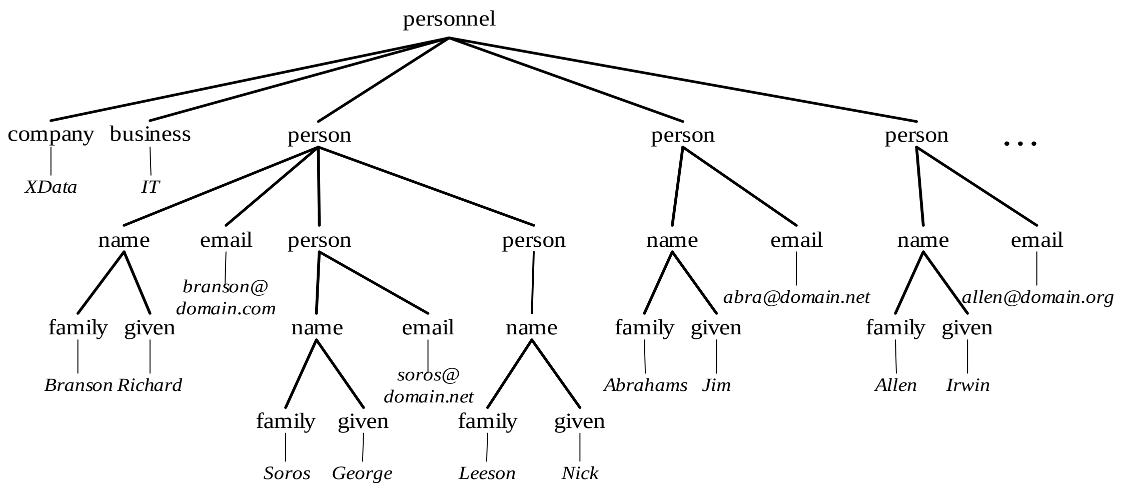


Figure 5.2. An XML example

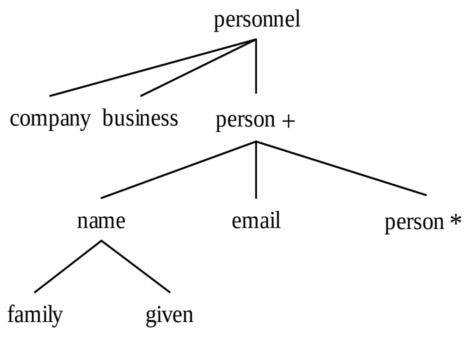


Figure 5.3. Element hierarchy

5.3 SCX: a structure coding

In this section, we describe the design of our novel structure coding SCX and the construction algorithms. For simplicity, we will use the notation of DTD. The main goals of the SCX design is the efficiency in XML query processing and the robustness in structural update.

The main component of SCX is the *structural identifier* that presents the schematic order of XML elements. The schematic order of an XML element is determined by the DTD element type declarations, in which the element participates. For example, if a tag name **b** appears in the element type declarations of the tag names **a** and **f** in a DTD then in any instance of the XML document conforming to the DTD, the tag name of the parent element of an element having the tag name **b** must be either **a** or **f**.

The structural identifier of a node uniquely determines both structural identifier as well as the tag name of the parent node using two *index function*s: the function parentSID returns the structural identifier of the parent node and the function nameID returns an integer value used together with the tag name of the child node to find the tag name of the parent node.

The function nameID is a novel and essential part of our method. Intuitively, each pair of the tag names of parent and child nodes in the DTD is mapped into an integer called the *child order*. The child order together with the tag name of the child node uniquely determines the tag name of the parent node. In other words, the following dependencies hold:

$$\texttt{parent\_tag}, \texttt{child\_tag} \longrightarrow \textit{child order} \tag{5.1}$$

$$\texttt{parent\_tag} \longleftarrow \texttt{child\_tag}, \textit{child order} \tag{5.2}$$

The benefit of the introduction of these functions is the ability to determine the tag names of all ancestor nodes of a node without the necessity to access to the secondary memory. Therefore, intermediate structural joins can be avoided and the indexing data of only the leaf nodes in the query tree structure is necessary for evaluating XML queries.

5.3.1 The description of SCX

SCX represents the schematic and actual orders of an XML node by the pair [*sid*, *ord*], where *sid* is the structural identifier and *ord* is a presentation of the node in an XML instance. There are several proposed presentations for the position of XML elements in an XML document, each of them uses the `docID` and `nodeLevel` together with one of the pairs (`preorder`, `postorder`), (`preorder`, `range`), or (`startPos`, `endPos`). The methods to determine the hierarchical relationship among XML nodes using these presentations are similar and can be converted from one to other with a minor modification. In this research, we adopt the 3-tuple (`docID`, `startPos`, `endPos`) to present XML nodes, i.e. the component *ord* of SCX. The `nodeLevel` parameter used in the prior presentations is omitted. The element ($\texttt{docID}$, $\texttt{startPos}_1$, $\texttt{endPos}_1$) is an ancestor of the element ($\texttt{docID}$, $\texttt{startPos}_2$, $\texttt{endPos}_2$) iff $\texttt{startPos}_1 < \texttt{startPos}_2$ and $\texttt{endPos}_2 < \texttt{endPos}_1$. Since the encoding is very useful in defining the preceding and following orders, the inclusion of the *ord* in SCX guarantees that SCX can help the evaluation of queries related to these orders. As will be shown in Section 5.4, *sid* and *ord* complement each other in query processing. Since the design of SCX helps to reduce the number of elements and attributes, the indexing data of which needed to be loaded for processing a query, the size of the combination makes a little impact on the performance of the technique.

Definition 8 *A c-group is the maximal group of the consecutive sibling nodes having the same tag name.*

The notion of *c*-group in the Definition 8 can be extended to encompass the sequence of the same subgroup of elements that appear multiple times. In a DTD element type declaration, a *c*-group corresponds to a single child element or a subgroup of elements, the cardinality of which is multiple, such as ‘`b`’, “`b*`”, or “`(c, d)*`”. We will discuss the decomposition of DTD in section 5.3.2 to deal with the extension. For simplicity, the current form of the Definition 8 is used.

Definition 9 *An enumeration of the nodes in an XML tree* **`T`** *is called “forehand” if the identifier of a node is smaller than the identifier of the siblings from the other c-groups to the right of the node in the same parent node.*

The forehand property guarantees that the identifiers of XML nodes reflect their order as described in DTD declarations.

Definition 10 *A function childOrd: (a,b) → Integer, that maps a pair of tag names a and b, where b appears in the element content of a in a DTD element type declaration, to an integer, is called "parent-name-determinable" if* $\neg\exists$ a_1 *and* a_2 *such that* $a_1 \neq a_2$ *but childOrd(*a_1*, b) = childOrd(*a_2*, b).*

In other words, the child order and the child tag name uniquely determine the parent tag name. The construction of the "parent-name-determinable" function childOrd from the declarations of a DTD will be described in section 5.3.2.

The structural identifiers of the nodes in an XML tree are generated in a preorder traversal by a forehand enumeration. All the nodes in a *c*-group are assigned the same integer equal to the sum of a basic value computed from the structural identifier of the parent node and the child order of the *c*-group computed by the parent-name-determinable function childOrd. The root node, itself is a *c*-group, is enumerated by 1.

Figure 5.4 shows a tree structure of the DTD declaration `<!ELEMENT a(b, c, d*, e)>`. The child nodes belong to four *c*-groups having the tag names `b`, `c`, `d`, and `e`. The nodes `d`s have the same *sid* equal to *l*+3.

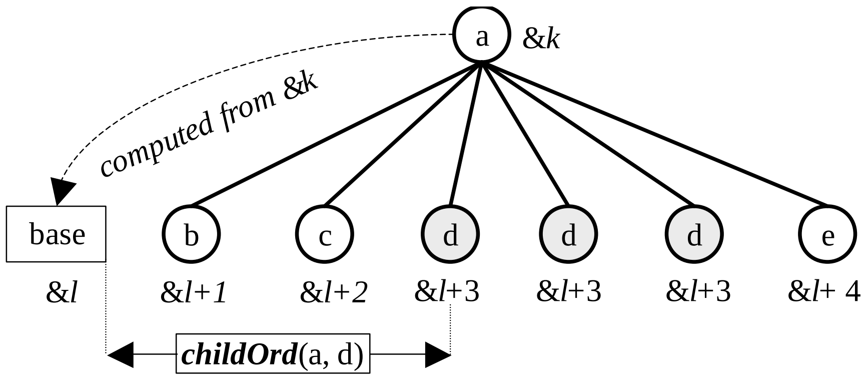


Figure 5.4. *c*-groups and structural identifiers

5.3.2 Generating SCX

Before describing the algorithm to generate SCX in section 5.3.2, we present the process to construct the parent-name-determinable function childOrd from DTD.

The construction of the childOrd function

The function childOrd returns an auxiliary integer used to distinguish the same child in different parents in a mapping from the child tag to the parent tag. The function is represented as a table constructed based on the DTD listing all possible arguments and values. In general, a DTD can be complex because of the complex specification of the type of an element. For example, an element **a** can be defined by a DTD declaration such as `<!ELEMENT a ((b, c, d)?, (e, f)*)>`. A direct transformation of such complex DTDs to the function childOrd is not the best solution.

A complete investigation to answer the question which types of DTD can enable the SCX is out of the scope of the current stage of the research. In this thesis, we propose a primitive solution to deal with the common-used types of DTDs. We decompose it using the *intermediate* elements recursively to reduce the complexity, where the intermediate elements are assigned unique tag names. The primary decomposition rules of DTD declarations are:

1. (b|c)* is decomposed into d* and d = (b|c)
2. (b, c)* is decomposed into d* and d = (b, c)
3. b*|c* is decomposed into d|e and d = b*, e = c*

The purpose of this decomposition is to guarantee the dependencies 5.1 and 5.2 by separating the occurrences of child elements with different tag names in the parent element. For example, the above complex declaration can be presented by an equivalent group of DTD declarations `<!ELEMENT g (b, c, d)>`, `<!ELEMENT h (e, f)>`, and `<!ELEMENT a (g?, h*)>`.

Note that the commonly found in practice DTDs do not contain such complex DTD declarations. For simplicity, we consider only the *meaningful* DTDs, i.e. any child element in actual data conforming the DTDs can be mapped uniquely to a child element in the DTD declaration of its parent element.

Since a DTD can be represented by a unranked tree automaton[16], its characteristics can be described by the language accepted by the automaton. Therefore, the SCX-enable DTDs can be also described by automata. Due to the relative dependence between the topic of the current research and the raised issue, we will discuss this issue in another work.

The core form of DTD declaration. In a DTD declaration, an element may have a number of sub-elements having the same tag name. The cardinality is omitted in construction of childOrd.

Definition 11 *The core form of a DTD declaration is received from the DTD declaration by replacing the cardinalities of the subelements by one.*

For example, the core form of the DTD declaration `<!ELEMENT a (b, c?, d*)>` is `<!ELEMENT a (b, c, d)>`.

Core child-orders. Each element is assigned a core child-order in its parent element equal to the index of the corresponding *c*-group.

Definition 12 *The index of a c-group is the order of the corresponding tag name in the core form of the DTD declaration of the parent element. The core child order of the node* **n** *in its parent node* **p***, denoted by* $\mathbf{p} \dashv \mathbf{n}$*, is equal to the index of the c-group containing* **n** *in* **p***.*

Note that the core child order is different from the actual order of child nodes and independent from the data size.

Example 9 *If an element* **a** *that conforms the DTD declaration* `<!ELEMENT a (b, c, d*, e)>` *has five child nodes* **d** *then the actual orders of the child nodes of* **a** *are:* **b**$\leftarrow$*1,* **c**$\leftarrow$*2,* **d[***1***]**$\leftarrow$*3,* **d[***2***]**$\leftarrow$*4,..,* **e**$\leftarrow$*8. The core child orders are:* **a** $\dashv$ **b**$\leftarrow$*1,* **a** $\dashv$ **c**$\leftarrow$*2,* **a** $\dashv$ **d[***1***]**$\leftarrow$*3,* **a** $\dashv$ **d[***2***]**$\leftarrow$*3,..,* **a** $\dashv$ **e**$\leftarrow$*4.*

Extended child-order. We extend the core child-order notion to guarantee the dependency (5.2). If $\exists$ $\mathbf{a}_1$ and $\mathbf{a}_2$ such that $\mathbf{a}_1 \neq \mathbf{a}_2$ but childOrd($\mathbf{a}_1$, **b**) = childOrd($\mathbf{a}_2$, **b**) then an integer value is added to the core child order of all the child nodes of $\mathbf{a}_2$ such that childOrd($\mathbf{a}_1$, **b**) $\neq$ childOrd($\mathbf{a}_2$, **b**). Since the cardinalities of XML documents are finite, the extended child-orders always exists.

Property 2. All values $e\mathscr{C}()$ are distinguishing.

The *extend child-order*s are stored in the table StruDTD that has three columns PAR, CHI, and cOrder containing the tag name of the parent elements, the tag name of the child elements, and the extended child-orders of the child elements, respectively. A row (a, b, o) of the table StruDTD means that any element having the tag name b can appear only in the o^{th} c-group of a parent element having the tag name a.

In Algorithm BuildStruDTD, a *segment* is a sequence of consecutive rows having the same value in the PAR column. The step 1 lists the pairs of parent-child tag names. For example, the DTD declaration `<!ELEMENT a (b*, c|d)>` corresponds to three rows in StruDTD, the columns PAR and CHI of which are **a** and **b**, **a** and **c**, **a** and **d**, respectively. Steps 2-7 compute the initial values of cOrder. Steps 8-12 generate the extended child orders. Steps 13 returns the table StruDTD and the fanout f.

Algorithm: BuildStruDTD

Input: A DTD or XML schema
Output: Table StruDTD and fanout f

```
1. save pairs a->b in PAR-CHI
/*initiating the possible value of the cOrder */
2. for each pair a->b
3.     if element content type is 'choice'
4.        $cOrder \leftarrow$ 1;
       else /*sequence*/
5.        if (a->b is the start of a segment)
6.           $cOrder \leftarrow$ 1;
7.        else $cOrder \leftarrow$previous $cOrder$ + 1; endif
       endif
   endfor
/* eliminating the duplication*/
8. loop
9.     for each pair a->b
10.       if $\exists$ a preceding pair a'->b &&
          $cOrder$s are equal then
11.          $\Uparrow$ $cOrder$ of a->* by 1
          endif
       endfor
12.    if all segments are fixed then break;
   endloop
13.returnPAR, CHI, $cOrder$, $f \leftarrow$max($cOrder$)
```

Note that this algorithm runs once with DTD or XML schemas, the size of which are independent from the size of the data sets associated with them. The table StruDTD is loaded in to main memory when the queries are processed. The intermediate elements resulted from decomposition of complex DTD declarations, if exist, also are included in the table StruDTD.

Example 10 *The table StruDTD of the element hierarchy in Figure 5.3 is shown in Table 5.2. The fanout f is equal to four. Both of the nodes* **`personnel`** *and* **`person`** *may have a node* **`person`** *as a child node, hence* **`personnel`** $\dashv$ **`person`** *must be different from* **`person`** $\dashv$ **`person`**.

PAR	CHI	cOrder
personnel	company	1
personnel	business	2
personnel	person	3
person	name	2
person	email	3
person	person	4
name	family	1
name	given	2

Table 5.2. A table StruDTD

Function childOrd. The function childOrd takes the values in the columns PAR and CHI in each line of the table StruDTD and returns the integer in the column cOrder. For example, in Table 5.2, childOrd(**person**, **name**) = 2. Since the dependency (5.2) holds, let parentTAG denote the function from (cOrder, CHI) to PAR, and we have

$$\mathsf{parentTAG}(\mathsf{childOrd}(\mathsf{a},\mathsf{b}),\mathsf{b}) = \mathsf{a} \tag{5.3}$$

Algorithm for SCX construction

Algorithm ConstructSCX generates SCX of the nodes of an XML document in a pre-order traversal. Set the fanout f equal to the maximal value of the column *cOrder* in

StruDTD that is independent from the size of XML documents. If **n** is a child node of **p** then the structural identifier of **n** is the sum of the base value equal to (**p**.sid -1) $\times$ f + 1 and childOrd(**n**.*tag*, **p**.*tag*).

Algorithm: ConstructSCX

```
Input:   T rooted at r, a fanout $f > 1$
Output:  SCX of nodes in T

1. travel T in the preorder
2.   if n is the root
3.     n.sid ← 1;
4.   else p ← parent(n);
5.     corder ← childOrd(p.tag, n.tag);
6.     n.sid ← f * (p.sid - 1) + 1 + corder;
     endif
7.   n.ord.startPos ← start position;
8.   n.ord.endPos ← end position;
   endtravel
```

The steps 5 and 6 of Algorithm ConstructSCX incorporate the child-orders computed by the function childOrd into the structural identifiers. The *sid* of the nodes of the XML document in Figure 5.2 are shown in Figure 5.5, where the fanout f is equal to four.

Example 11 *In Figure 5.5, suppose we have to generate the sid of the node* **name** *of the first node* **person**. *Since the code is generated in a preorder traversal, the sid of the parent node* **person** *is already known to be equal to four. The function nameID(***person**, **name***) = 2. Therefore, the sid of the node* **name** *is equal to* $4 \times (4 - 1) + 1 + 2$*, or 15.*

5.3.3 Index functions

The index functions are used to navigate between a node and its parent node based on the structural identifier of SCX. Given the fanout f and a node **n**, the function

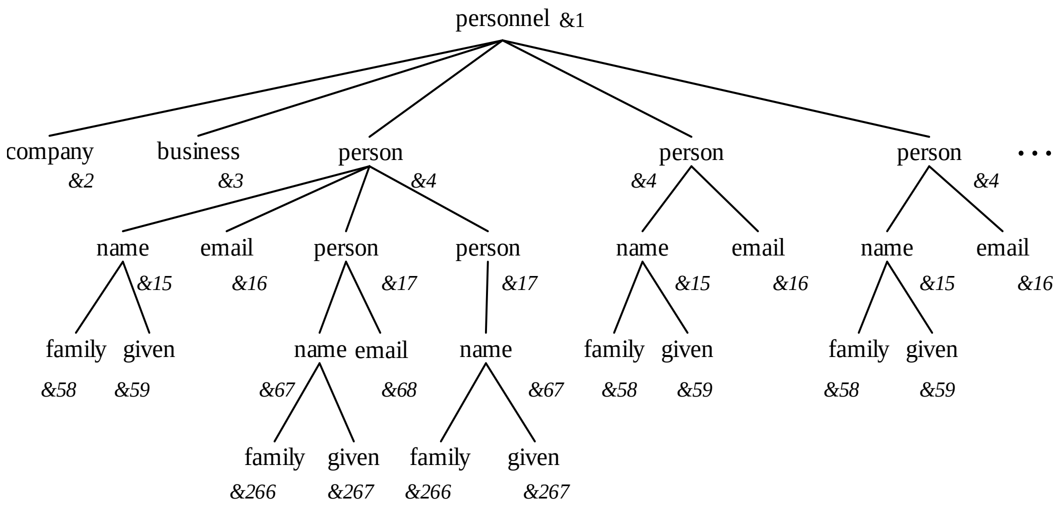


Figure 5.5. *sid* of the document in Figure 5.2.

parentSID is defined as the following:

$$\mathsf{parentSID}(\mathbf{n}.sid) = \lfloor(\mathbf{n}.sid - 2)/f\rfloor + 1 \tag{5.4}$$

The purpose of the function parentSID is to compute the structural identifier of the parent node. This property of SCX is similar to the enumeration introduced in [52]. However, our structural identifier system is more robust and realistic. In addition, the introduction of the function nameID makes SCX more efficient than the method in [52]. The function nameID is defined as the following:

$$\mathsf{nameID}(\mathbf{n}.sid) = \mathbf{n}.sid - f \times \lfloor(\mathbf{n}.sid - 2)/f\rfloor - 1 \tag{5.5}$$

This function returns the child order of **n** in its parent node. In other words, it computes the *distance* from the start of the interval of possible structural identifiers for the child nodes to the structural identifier of **n**.

Lemma 2 *If the structural identifiers of SCX are generated by Algorithm ConstructSCX then given a node, the structural identifier and the tag name of the parent node of the node can be determined.*

Proof: Let **n** and **p** denote a node and its parent node. We shall show

$$\mathbf{p}.sid = \mathbf{n}.sid - f \times \lfloor(\mathbf{n}.sid - 2)/f\rfloor - 1 \tag{5.6}$$

From the steps 5 and 6 of Algorithm ConstructSCX, we have

$$\mathbf{n}.sid = f \times (\mathbf{p}.sid - 1) + 1 + \mathsf{childOrd}(\mathbf{p}.tag, \mathbf{n}.tag) \tag{5.7}$$

Since $1 \leq$ childOrd(**p**.tag, **n**.tag) $\leq f$,

$$f \times (\mathbf{p}.sid - 1) + 2 \leq \mathbf{n}.sid \leq f \times (\mathbf{p}.sid - 1) + 1 + f \tag{5.8}$$

or

$$f \times (\mathbf{p}.sid - 1) \leq \mathbf{n}.sid - 2 < f \times (\mathbf{p}.sid) \tag{5.9}$$

Divide to $f > 0$, we have

$$\mathbf{p}.sid - 1 \leq (\mathbf{n}.sid - 2)/f < \mathbf{p}.sid \tag{5.10}$$

Therefore,

$$\mathbf{p}.sid = \lfloor(\mathbf{n}.sid - 2)/f\rfloor + 1 \tag{5.11}$$

Thus, (5.6) holds and the structural identifier of **p** can be computed by Formula 5.4. Furthermore, to determine the tag name of **p**, from (5.11) and $f > 0$, we have

$$f \times \lfloor(\mathbf{n}.sid - 2)/f\rfloor = f \times (\mathbf{p}.sid - 1) \tag{5.12}$$

or

$$\mathbf{n}.sid - f \times \lfloor\frac{\mathbf{n}.sid - 2}{f}\rfloor - 1 = \mathbf{n}.sid - f \times (\mathbf{p}.sid - 1) - 1 \tag{5.13}$$

From (5.5), (5.7), and (5.13), we have

$$nameID(\mathbf{n}.sid) = \mathsf{childOrd}(\mathbf{p}.tag, \mathbf{n}.tag) \tag{5.14}$$

From (5.3) and (5.14),

$$\mathbf{p}.tag = parentTAG(nameID(\mathbf{n}.sid), \mathbf{n}.tag) \tag{5.15}$$

The lemma holds. □

Note that the determination of the *sid* and the tag name of the parent node is independent from the *ord* of the child node.

Example 12 *In Figure 5.5, for the node* **`given`**, *the sid of which is equal 267, the sid of the parent node is equal to* $\lfloor(267 - 2) / 4 \rfloor + 1$, *which is equal to 67. The functions nameID returns* $267 - 4 \times \lfloor(267-2)/4\rfloor - 1$, *which is equal to 2. The function parentTAG(***`given`***, 2) returns the value* **`name`**. *Similarly, the tag names of the ancestor nodes are found to be* **`person`**, **`person`**, *and* **`personnel`**, *respectively. Therefore, the full node path for the node* **`given`** *is "`personnel/person/person/name/given`".*

5.3.4 Other features of SCX

Coding complexity

We generated SCX for an XML document using the SAX parser [4]. A stack, the size of which is the maximal height of the XML tree, keeps the current node path. Two other buffers keep the tag names and the *sid*s of the previous nodes. A node is visited after the *sid* of its parent node is known. The cost of the function childOrd is log_2(size(StruDTD)). Therefore, the cost for generating SCX is $O(log_2$(size(StruDTD)) $\times$ (data size)). In our experiments, it took 90 seconds for generating in the main memory the SCX of a data set of 4103211 elements and attributes.

Coding size

The interesting feature of the SCX is that the fanout used to compute *sid* depends on the number of *c*-groups rather than the degree of the nodes in the XML instances. For example, according to the DTD declaration `<!ELEMENT a (b?, c*, d)>`, the degree of **`a`** increases in parallel to the number of **`c`**. However, the number of *c*-groups is 3, a fixed value. The small size of the fanout keeps the size of SCX small. In Table 5.3, we provide the sizes of the data sets used in our experiments and of the SCX indexing data (in the column **StrInxSize**) generated from the data sets as an illustration.

Robustness of SCX

The structural identifier of SCX is robust for the structural update. Taking into account the DTD or XML schema, SCX *anticipates* the position of updated nodes in the associated XML documents. The structural updates that significantly affect the robustness of the numbering schemes are:

1. **The increase of the number of nodes having the same tag** in a parent node. For example, according to the DTD declaration `<!ELEMENT a (b?, c*, d)>`, a node having the tag name **`a`** may have a number of nodes having the tag name **`c`**. All these nodes **`c`** have the same *sid*. A new node **`c`** must belong to the *c*-group of existing **`c`**, hence it has the same *sid*. Furthermore, it is always possible to select the **`c`**.*ord* such that the new **`c`** will be reside in the correct order. The order of the new **`c`** among the nodes **`c`***s* is guaranteed by an appropriate value of the *ord* and does not affect the selection of the structural part *sid* of the new node.

2. **The uncertain occurrence** of an element in its parent element: Let consider the same DTD declaration `<!ELEMENT a (b?, c?, d)>`. A node having the tag name **`a`** may or may not have a child node having the tag **`b`**. According to the construction of SCX, a place for such a **`b`** is reserved.

Therefore, in both cases, the insertion of a new node does not cause the change of the *sid* of the other nodes. Note that SCX is robust in the terms of structural update with *clue*, i.e. the schematic information from DTD or XML schema is available.

Coping with changing DTD

A radical change of DTD, which leads to the entire change of document content and structure, requires rebuilding the index from scratch. The insertion of an element in the content specification of the DTD declaration of an existing element may change the SCX of related elements in the actual data. If insertions are predicted then a *sparse* mode of SCX construction can reserve the *location* for the elements to be inserted. If an inserted element is the last child element of its parent element and does not increase the maximal value of *cOrder* then SCX of existing data is not changed. In practice, the changes in a DTD are rarer than the changes in the content and structure of the XML documents conforming to the DTD.

5.4 Virtual Joins with SCX

SCX provides the Virtual Joining mechanism that can avoid many the intermediate structural joins in XML query processing. To perform Virtual Join mechanism, we need several basic functions.

Determining the tag name of the parent node. For a given node, the function findParentSidAndTag calls the parentSID and nameID functions to compute the *sid* of the parent node and the child order, which is used by the function parentTAG to find the tag name of the parent node.

```
Function:   findParentSidAndTag

Input:  n.sid, n.name
Global:  the fanout f of T

1. sid←parentSID(n.sid);
2. tid←nameID(n.sid);
3. name←parentTAG(n.name, tid);
4. return sid, name;
```

Establishing node path. The function generateNodePath establishes the full node path for a given node by recursively performing the function findParentSidAndTag.

Checking the existence of an ancestor having a specific tag name. For a given node with its *isd* and tag name, the function findAncByName look for the lowest ancestor that has a given name.

```
Function:   findAncByName

Input:  n.sid, n.name, ancname
Global:  the fanout f of T

1. name←n.name, sid←n.sid;
2. loop(sid >1)
3.   sid, o ←findParentSidAndTag(sid, name);
4.   name←parentTAG(name, o);
5.   if (name is empty) return null;
6.      else if (name = ancname) return sid, name;
   endloop
```

The function findAncByName can be modified by replacing the step 6 with the following command:

`if(`*name* `=` *ancname* `&&((`$l>$`0 &&` i`=`l`)||` l`=0))`
to incorporate the level of the ancestor to be looked for. It checks if the *l*-level ancestor has a given tag name. If *l* is equal to 0, the level requirement is omitted.

5.4.1 Basic *path-predicate* queries

In this section, we describe how to apply the Virtual Join mechanism to process the queries represented by a path expression ended with a predicate.

Definition 13 *A path query is called basic path-predicate query if it is expressed in the form:* $\mathsf{a}_1\ell_1\mathsf{a}_2\ell_2\cdots\ell_{k-1}\mathsf{a}_k$ *or* $\mathsf{a}_1\ell_1\mathsf{a}_2\ell_2\cdots\ell_{k-1}[\mathscr{P}$ *of* $\mathsf{a}_k]$*, where* $k \geq 1$*,* ℓ_i *(i = 1 to k-1) is either the parent axis '/' or the ancestor axis '//', and* $\mathscr{P}$ *is a predicate of* a_k*.*

For example, "`person/name[given = 'Smith']`" is a basic path-predicate query. In the basic path-predicate queries, all the nodes having the tag names a_i, $i \neq k$, and the nodes having the tag name a_k filtered by the predicate $\mathscr{P}$ participate in the structural joins. The predicate $\mathscr{P}$ is optional and can be void.

A basic path-predicate query is presented by a table called *query pattern* with four columns. The column TAG contains the tag names in the query path expression in reverse order, i.e. a_i, (i = k-1 to 1). The column AXIS contains the ℓ_i (i = k-1 to 1). The column ANS indicate the lines having the axis '//' in the column AXIS. In the column FROM, only the values of the lines having the axis '//' in the AXIS column are used and initially equal to 0.

The function matchPattern checks if a node having the tag name a_k matches a query pattern. The function step(i) looks for the element having the tag name TAG[i] in the axis AXIS[i] of the current node by calling the function findAncByName. If the AXIS[i] is '//' and FROM[i] $>$ 0 then *sid* of the node to be found must be less than FROM[i]. The function step returns *null* if there is no such a node, otherwise returns the found *sid* and tag name. The step 11 seeks the next possible root for a sub-path starting by '//'. The function matchPattern returns *true* if all the steps in the query pattern are satisfied.

Function: matchPattern

```
Input:  a node $\mathbf{n}$, a $query\ pattern$
1. $cursid$ ←$a_k$.$sid$; $curtag$ ←$a_k$.$tag$; $curstep$ ←1;
2. while ($true$)
3.   $b$ ← step($curstep$)
4.   if ($b$!=$null$)
5.     $cursid$ = $b$.$sid$; $curtag$ = $b$.$tag$;
6.     if (AXIS[$curstep$]='//')
7.       FROM[$curstep$] = $b$.$sid$;
       endif
8.     if $curstep$ = sizeOf(querypattern)
9.       return $true$;
10.    else $curstep$++;
     else
11.    Seek max $j < curstep$:  AXIS[$j$]='//'&& FROM[j] > 1
12.    if ∃:  $curstep$ ← $j$; ∀ i>j FROM[i] ← 0;
13.    else break;
     endif
   endwhile;
14.return $false$;

Function step(i)
1. $s$ ← $cursid$; $t$ ← $curtag$;
2. $\mathbf{n}$←findAncByName($s$, $t$, TAG[$i$], AXIS[i]);
3. return $\mathbf{n}$;
```

In function matchpatter, the function generateNodePath can be applied to avoid the repetition of the function step.

Lemma 3 *The function matchPattern correctly verifies if the element a_k satisfies the path expression $a_1\ell_1 a_2\ell_2\cdots\ell_{k-1}a_k$.*

Performing Virtual Joins. The procedure VirtualJoin takes a set of the nodes $\mathbf{a}_k$, which satisfy the predicate $\mathscr{P}$, and a query pattern as the input. By a single scan over the set, for each index *i*, the function matchPattern checks if the node $\mathbf{a}_k^i$ matches the query pattern. Note that the function matchPattern will terminate early for the disqualified nodes and the checking process run totally in main memory. The function virtualJoin can process the *ancestor-level* joins, where the hierarchy level is required, such as "`a/b`", "`a/*/b`", as well as the "`a//b`".

```
Function:  VirtualJoin  /*for basic path-predicate query*/

Input:  dlist[], queryPattern

1. for (i from 0 to the size of dlist - 1)
2.  if matchPattern(dlist[i], pattern) = true
3.     output dlist[i];
4.  endif;
   endfor;
```

The Virtual Join mechanism does not require the candidate nodes to be sorted and evaluates basic path-predicate queries without I/O except the indexing data of the output candidate set. For queries with long location paths, as shown by the experiment results, the Virtual Join mechanism has a clear advantage since the indexing data of only the last elements in the location paths is needed to be loaded.

5.4.2 Complex *path-predicate* queries

A path query may be associated with several selection predicates.

Definition 14 *A path query is called complex path-predicate query if it is expressed in the form of a finite sequence of basic path-predicate queries separated by the parent axis '/' or the ancestor axis '//'.*

A complex path-predicate query $\mathsf{B}_1\ell_1\mathsf{B}_2\ell_2\cdots\ell_{k-1}\mathsf{B}_k$, $k \geq 1$, is evaluated by integrating the result of the basic path-predicate queries B_i, i = 1 to k, which are evaluated separately using the Virtual Joins. Let $\{\mathbf{r}_i\}$ denote the list of result nodes of B_i, $\{s_i\}$ denote the list of *sid* of the nodes in the highest hierarchical structure of B_i corresponding to $\{\mathbf{r}_i\}$. From the description of matchPattern, $\{s_i\}$ is generated together with $\{\mathbf{r}_i\}$.

The lists $\{\mathtt{r}_i\}$ are joined using the conventional structural join technique to produce the final result. Two elements $\mathtt{r}_i^j$ and $\mathtt{r}_{i+1}^k$, $1 \le i < k-1$, are matched in the structural joins of $\{\mathtt{r}_i\}$ and $\{\mathtt{r}_{i+1}\}$ if:

1. $\mathtt{r}_i^j.startPos < \mathtt{r}_{i+1}^k.startPos$ && $\mathtt{r}_{i+1}^j.endPos < \mathtt{r}_i^k.endPos$, and
2. (ℓ_i is '/' && $\mathtt{r}_i^j.sid$ = parentSID(s_i^j))||(ℓ_i is '//' && $\mathtt{r}_i^j.sid < s_i^j$)

For example, the complex *path-predicate* query `a/b/c/[d:` $\mathscr{P}_1$`]/e/f/[g:` $\mathscr{P}_2$`]` is decomposed into two subqueries `a/b/c/[d:` $\mathscr{P}_1$`]` and `/e/f/[g:` $\mathscr{P}_2$`]`. For the first subquery, the **d** satisfying the predicate $\mathscr{P}_1$ are loaded and virtually joined with **c**, **b**, **a**. For the second subquery, the **g** satisfied the predicate $\mathscr{P}_2$ are loaded and virtually joined with **f** and **e**. The *sid* of the nodes **e** corresponding to the intermediate nodes **g** are also available. These **d** and **g** are joined by the condition:"**d** is an ancestor of **g** and **d**.*sid* = parentSID(**e**.*sid*), where the **e** corresponds to the **g**".

5.4.3 Processing twig queries

Queries on XML data typically specify elements by selection predicates and their tree structure relationship that can be represented as a node label twig pattern with elements with or without predicates in the leaf nodes.

A twig query is decomposed into three complex path-predicate subqueries that are processed separately using the Virtual Joins. The result elements of these queries then are joined using the component *ord* of SCX by conventional structural join techniques that base on the (`startPos`, `endPos`) presentation of the position of elements, e.g. [35], [27] etc. The compatibility with the prior researches is an interesting feature of SCX.

Figure 5.6 illustrates a twig query, the result elements of which have the tag name `b`. The twig query is decomposed into three subqueries represented by the paths from **a** to **b**, from the node bellow **b** in the left branch to **p**, and from the node bellow **b** in the right branch to **q**. The join of the outputs of these subqueries to produce the final answer is similar to the join of intermediate results in a complex path-predicate query.

Example 13 *Using SCX, the XQuery statement "`FOR $b IN /site/people/person WHERE $b/address/city = 'Nara'] RETURN $b/name/text`" is decomposed*

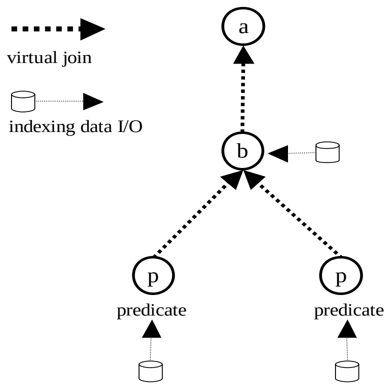


Figure 5.6. Processing a twig query.

into the basic path-predicate queries "`/site/people/person`", "`address/city = 'Nara']`", and "`name/text`".

Note that, in general, the I/O complexity is *optimal* since only the indexing data of elements that have to be verified by predicates is loaded or belong to the candidate set of the output. For example, in Example 13, the indexing data of elements **`person`**, **`city`** and **`text`** are needed to perform the joins.

DSet	**Size**	**#ele.**	**#att.**	**StrInxSize**
1	23.4MB	336224	76867	23MB
2	57.6MB	832911	191162	55MB
3	87.2MB	1253793	286576	87MB
4	115.7MB	1666315	381878	110MB
5	145.1MB	2088879	478374	148MB
6	174.0MB	2502484	573122	158MB
7	203.4MB	2921324	669773	188MB
8	232.2MB	3337649	765562	219MB

Table 5.3. Specifications of the data sets

5.5 Experiment

We have implemented SCX and the Virtual Join mechanism in a system called Virtual Joins Engine for XML(VJEX). The current version of VJEX has the module for evaluating the basic path-predicate queries. We maintain the main file structure for storing the structure coding as the following:

```
<scx, element_name, add_infor>
```

where `scx` is the SCX, including both *sid* and *ord*, `element_name` is the tag name, and `add_infor` consists of an indicator whether the item is an element or attribute and a pointer to data. The primary functions on the structures are:

F_1: For a given `scx`, retrieve the elements or attributes having that identifier.

F_2: For a given name, retrieve all the elements or attributes having that name.

The XML content and the coding data are indexed using B^+-tree. We create two B^+-trees on `scx` and `element_name`. Since it is not necessary to sort the indexing data in Virtual Joins, the data is sorted by the `scx.ord.startPos` as required by most of the structural join techniques. For the purpose of this research, the physical data presentation of the XML content is not material and its details is not relevant to the results of this research.

VJEX keeps the table StruDTD in the main memory during the query evaluation. The input query of VJEX is transformed into a basic path-predicate, a complex path-predicate, or a twig forms. The basic path-predicate subqueries are evaluated separately using the Virtual Join mechanism. The partial results are joined using non-virtual join algorithms (not shown here) to produce the final result.

5.5.1 Experiment setup

We measure the *full* processing time that includes the elapsed times for loading the indexing data and for performing structural joins in XML queries.

We compare our method with the method Stack-Tree-Decs in [35] and PathStack in [27]. Both of these Algorithms use the tuple (`docID`, `startPos`, `endPos`, `nodeLevel`) to present an XML element. Stack-Tree-Decs is the highest performance method among four methods described in [35], where stacks are used to reduce the number of the match tests for the pairs of elements from the joined candidate sets in structural joins. PathStack also uses stacks to compactly present the partial and total answers to avoid the large intermediate answers.

5.5.2 Experimental platform and Data sets

Our experiments were conducted in a workstation running on Windows XP Professional with a 2GHz CPU. The SAX parser available from the Xerces project [4] were used to parse XML data. Other programs for extracting structure information from DTD, generating SCX, and processing the Virtual Joins were written in Java. The maximal Java heap size was set to be equal 300MB.

We used the XML data generator xmlgen provided by XMark [2] to generate synthetic XML documents conforming the DTD "auction.dtd". This DTD has a fairly complex structure to make the experiments objective. The specifications of these data sets are shown in Table 5.3. As shown in the column **StrInxSize** of Table 5.3, the size of the SCX indexing data is approximately equal to the size of the XMark data sets from which it was generated.

5.5.3 Experimental results

To evaluate SCX, we use the queries that features the various complexities of structural joins, some of them were borrowed from prior researches in the same topic. The queries contain both '/' and '//' axes and include short, medium, and long location paths as shown in Table 5.4.

The experiment results are shown in Figures 5.7, 5.8, and 5.9, where the methods SCX, Stack-Tree-Decs, and PathStack are abbreviated by SCX, STD and PS, respectively. The number of hits of the queries are shown by the columns **Q1**, **Q2**, **Q3**, **Q4**, **Q5**, and **Q6** in Table 5.5 according to the data sets.

Simple joins

The queries with a single structural join have been discussed in detail in the [31] such as parent-child joins "`E1/E2`" or ancestor-descendant joins "`E1//E2`". We omit the discussion about the element and attribute join, considering it as a special case of

Queries	XPath expressions
Q_1	`//closed_auction/item`
Q_2	`//items/name`
Q_3	`//open_auction//description`
Q_4	`//open_auction//description//listitem`
Q_5	`//open_auction//description//keyword`
Q_6	`//closed_auctions/closed_auction/annotation/description/ parlist/listitem/text/emph/keyword/`

Table 5.4. Query set for testing Virtual Joins

"`E1/E2`" join. Both of the queries Q_1 and Q_2 involve a single parent-child join of two element sets. The query Q_3 has an ancestor-descendant join. The cardinalities of the XML element sets participating in the structural joins are different. From the Figure 5.7(a), we can see the SCX is slightly better than Stack-Tree-Decs in the smallest data set and significantly better in the bigger data sets. The elapsed times for processing the short queries on different data sets are shown in Figure 5.7(a)(b) and 5.8(a).

Medium complex queries

The queries contain several structural joins that may be ancestor-descendant or parent-child joins. The queries Q_4 and Q_5 are borrowed from [40]. In the queries Q_4 and Q_5, there are only the ancestor-descendant joins. The elapsed times for processing the queries on different data sets are shown in Figure 5.8(b) and 5.9(a). As expected, the axis '//' can be processed efficiently using Virtual Joins. For the location paths of Q_4 and Q_5, the indexing data of only two elements **`listitem`** and **`keyword`** was loaded and the remaining part of the evaluation process was done in main memory.

Complex queries

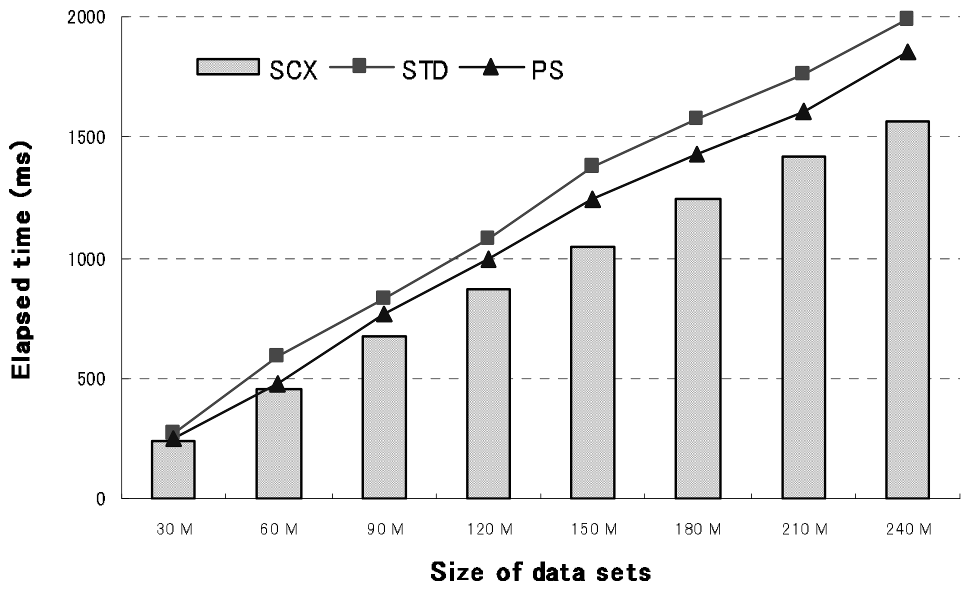
A number of queries introduced in XMark [2] have complex structure. The evaluation of such queries requires many structural joins. An example of such queries is Q_6.

Performing the structural joins in **Q6** is the main workload of the evaluating the query: "*Print the keywords in emphasis in annotations of closed auctions*". The elapsed time for processing the query on different data sets is shown in Figure 5.9(b). For the complex query, the advantage of the SCX over STD and PS is greatly significant. It can be explained by that the amount of indexing data saved by SCX from loading from secondary memory, that is required by other indexing methods, is larger for such queries. For the location path of Q_6, the indexing data of the only element **`keyword`** was loaded and the remaining part of the evaluation process was done in main memory despite of the number of joins in the location path.

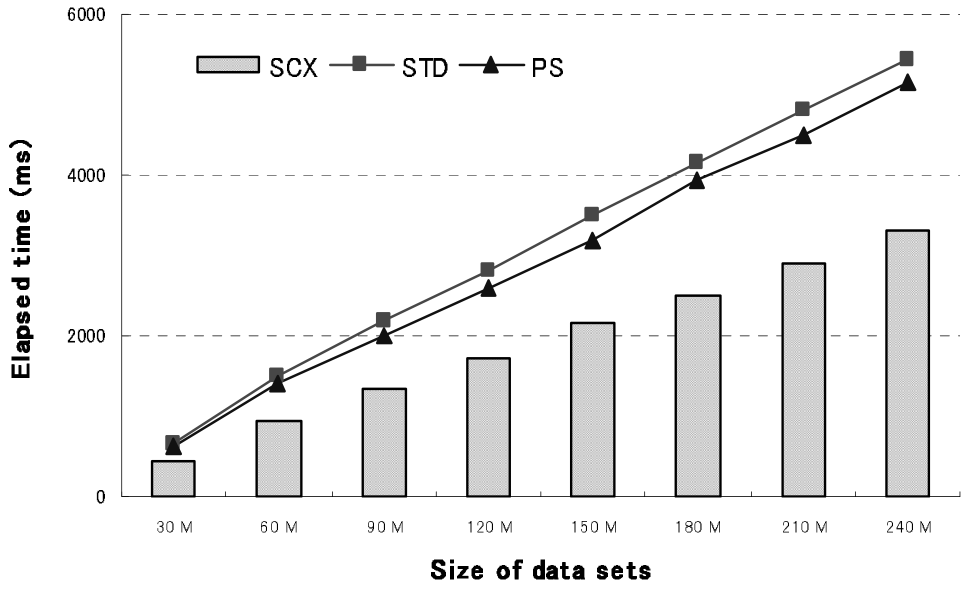
DSet	Q1	Q2	Q3	Q4	Q5	Q6
1	1952	7350	2400	3528	3173	50
2	4875	10875	6000	8361	7471	134
3	7312	16312	9000	12759	11341	203
4	9750	21750	12000	16640	14954	271
5	12187	27187	15000	21123	19030	318
6	14625	32625	18000	25499	22917	412
7	17062	38062	21000	29706	26551	448
8	19500	43500	24000	33954	30377	521

Table 5.5. Numbers of hits of the test queries

In our query set, the join workloads are increased from queries Q_1 to Q_6. In all experiments, we can see an interesting tendency that when the sizes of data sets increase, the comparison result is changed in the favor of the SCX method. The advantage of SCX steadily increases in correspondence with the size of the experimental data sets as well as the join workload. The experiment result accords with our expectation that for the large data sets and the queries with the heavy join workload, the advantage SCX in I/O complexity for XML query processing becomes more significant since the amount of data saved from I/O becomes larger.

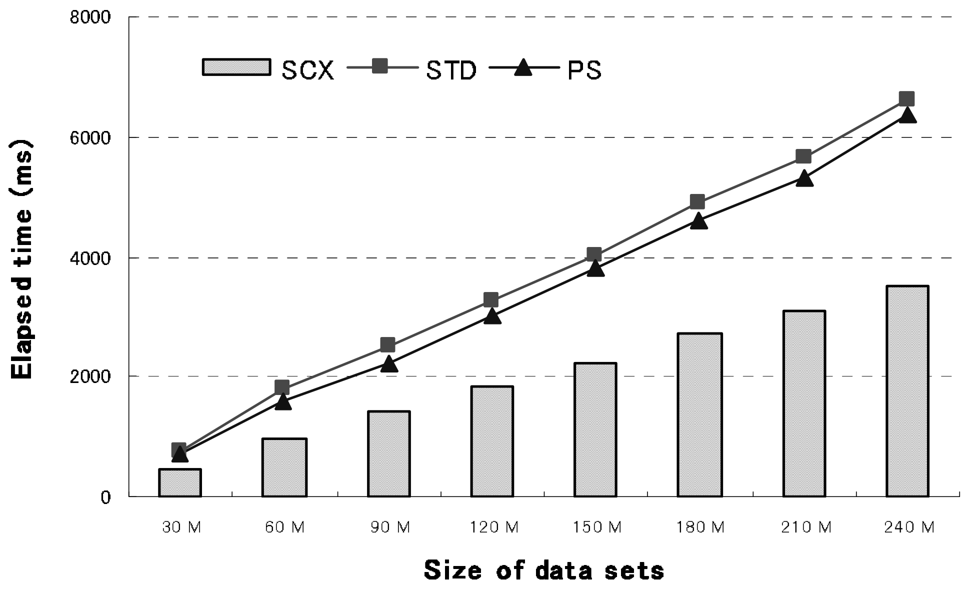


(a) Elapsed query time for Q_1

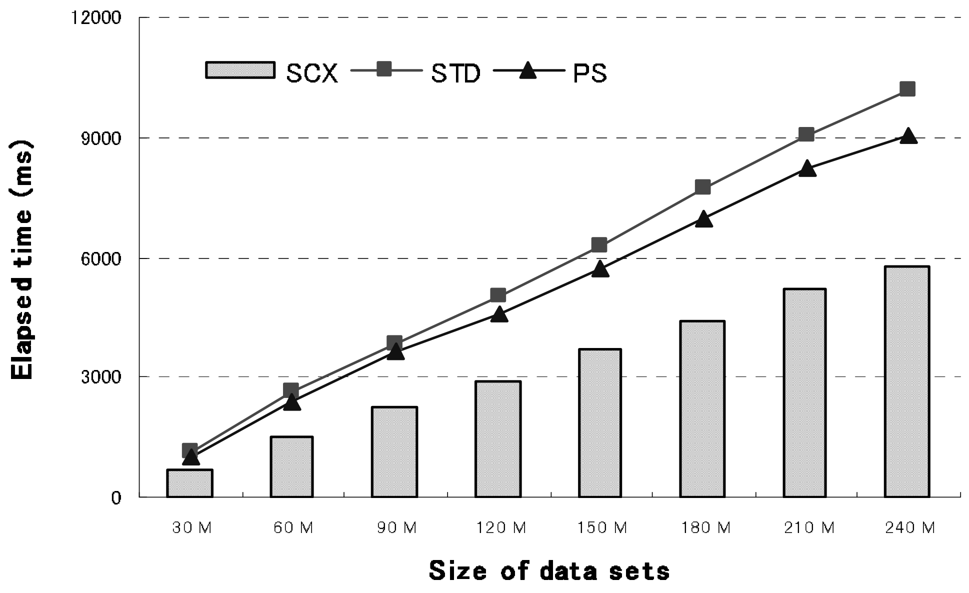


(b) Elapsed query time for Q_2

Figure 5.7. Query processing of the short queries Q_1 and Q_2.

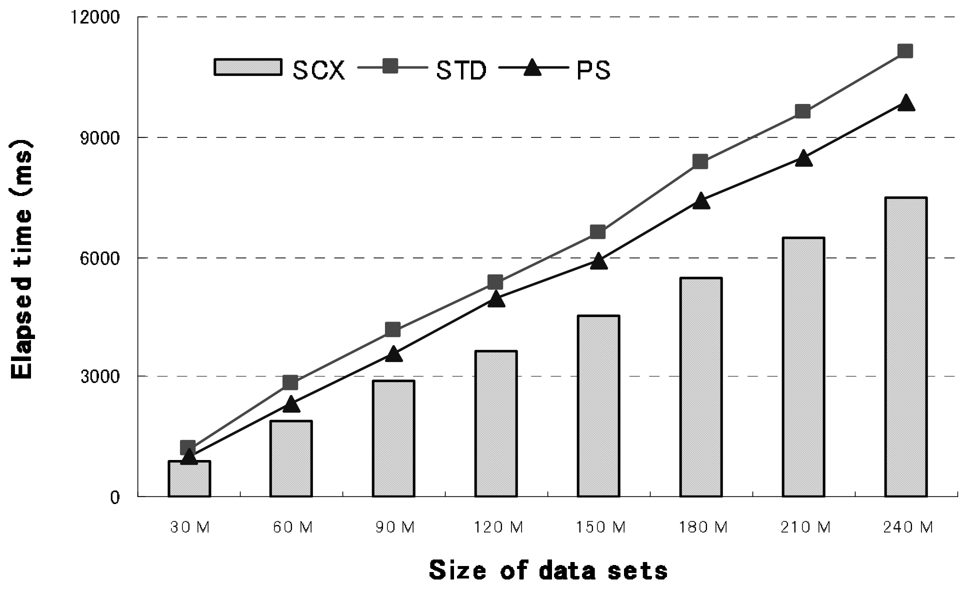


(a) Elapsed query time for Q_3

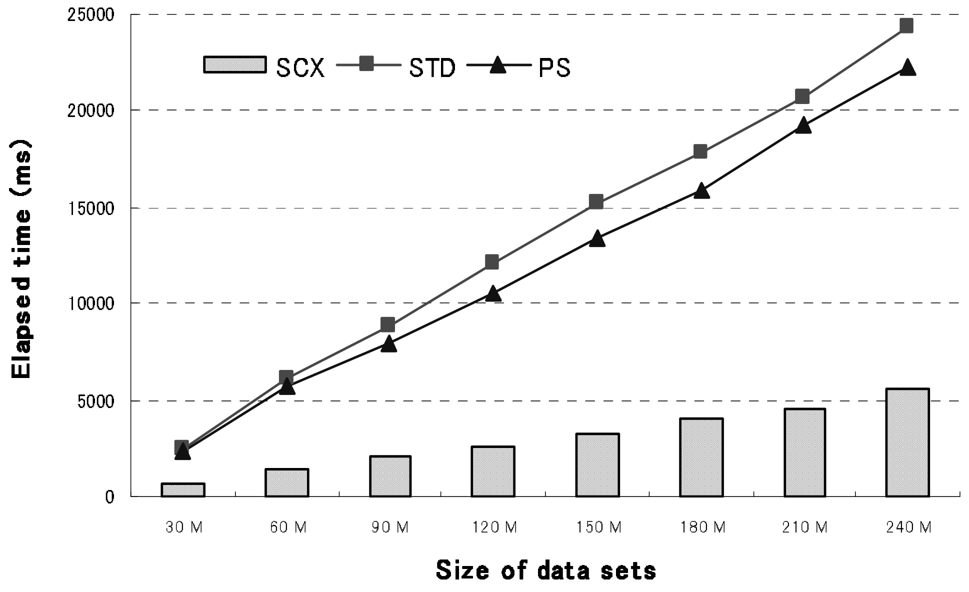


(b) Elapsed query time for Q_4

Figure 5.8. Query processing of the short query Q_3 and the medium query Q_4



(a) Elapsed query time for Q_5



(b) Elapsed query time for Q_6

Figure 5.9. Query processing of the medium query Q_5 and the complex query Q_6

5.6 Summary of Chapter 5

Prior techniques for processing XML queries have not utilized efficiently the document structure described in DTD and XML schema and heavily depended on the actual order of XML elements in XML instances. In this study, we proposed the Virtual Join mechanism for structural joins using the Structure Coding for XML data (SCX) that incorporates both structure and tag name information extracted from DTD or XML schema. SCX greatly improves the I/O complexity of XML query processing. Many intermediate containment joins can be avoided by computing the result set using the operations on the SCX only. According to our experiments, SCX significantly improves the query processing efficiency in correspondence with the structural join workload and the size of data sets. In addition, SCX and prior structural joins techniques can be integrated to improve the XML query processing.

Chapter 6

Summary and Future Research

6.1 Conclusion

Since XML has been widely considered as a leading candidate for the data organizing principle in Internet, the amount of data formatted in XML expands quickly. Therefore, the need for efficient indexing structures for XML data is more and more significant for XML data management.

In this thesis, we present the result of the studies on efficient XML indexing structures. We propose three indexing structures for XML data to address three common issues of XML data management as follows:

1. We have founded that to cope with extensively content-updated XML data, the expression of the position of elements in XML documents by Relative Region Coordinate is a solution. RRC can reduces the workload to maintain the coordinate integrity for XML data in XML indexing structure.

2. We have proposed the novel recursive numbering scheme *r*UID that is robust in structural update, scalable for coding arbitrarily large XML documents, and expressive in presenting the main XPath axes. The indexing structure based on the numbering scheme can deal well with the XML, where the new XML elements can be inserted without clues, i.e. DTD or XML schemas.

3. We have showed that the schematic information extracted from DTD or XML schemas can be used effectively to improve the efficiency of XML query pro-

cessing. For this purpose, we have proposed a mechanism called Virtual Joins that allows greatly reduce the I/O workload needed for processing XML queries. The mechanism is based on a structure coding for XML data called SCX that is robust on the structural update of the XML documents conforming DTD or XML schemas.

Design of efficient XML indexing structures is a complex task because XML's semantic characteristics effect each other. Our solutions presented in the thesis is the result of exhausted analysis, extensive testing and performance evaluation. Although some parts of the thesis can be developed further, based on the experimental results, we concluded that the proposed indexing structures are competitive in solving the specific issues that the structures have been desinged to aim at.

6.2 Future Research

A further development of the research issues addressed in the thesis is likely going to improve or develop some additional properties of the proposed indexing structures. For example, an extension of Chapter 4 is to investigate the impact of the *r*UID frame on the effectiveness of query processing. The variants of the indexing structures *r*UID and SCX specific for various platforms are in our next plan. Currently, we are going to investigate the application of Virtual Joins in the applications categorized into the *XML-Enabled Database* group, as discussed in Section 2.3.1, page 23 of this thesis.

Following a comment from the Thesis Committee, it is worth to investigate in detail which types of DTD enable the SCX. In this thesis, a primitive solution has been proposed to deal with the common-used types of DTDs. For a comprehensive investigation, a tentative approach is using the automata theory to represent DTDs and providing the analysis of the characteristics of SCX-enable DTD based on the richness of the automata theory. A short discussion has been added to Chapter 5 to outline the approach. However, due to the theoretical nature of the approach, it would better to be discussed in a new work.

Another interesting direction is to investigate the functional characteristics of these structures in new application domains. Since XML is the next wave of the Internet technology, the research domain related with XML covers a large range of applications such as stream data management, e-commerce applications, XML data management for bio-informatics, etc. However, due to the differences of the types of data generated in the applications, it is likely that each application requires further refinement and improvement of the current proposed techniques, including indexing structures, for XML data management.

References

[1] A. Biliris. An Efficient Database Storage Structure for Large Dynamic Objects. *Proceedings of the International Conference on Data Engineering, Arizona, USA*, pages 301–308, 1992.

[2] A. Schmidt. XMark: A Benchmark for XML Data Management. *Proceedings of the International Conference on Very Large Databases, Hongkong, China*, 2002.

[3] Akmal B. Chaudhri, Awais Rashid, Roberto Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison Wesley, ISBN: 0201844524, 2003.

[4] Apache Software Foundation. Apache XML Project. *http://xml.apache.org/*, 2001.

[5] B. Cooper, N. Sample, M. Franklin, G. Hjaltason, M. Shadmon. A Fast Index for Semistructured Data. *Proceedings of the International Conference on Very Large Databases, Hongkong, China*, pages 341–350, 2001.

[6] C. C. Kanne, G. Moerkotte. Efficient Storage of XML Data. *Proceedings of the International Conference on Data Engineering, California, USA*, page 198, 2000.

[7] C. F. Goldfarb, P. Prescod. *XML Handbook, 4th edition*. Prentice Hall, New Jercey, USA, 2002.

[8] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. *Proceedings of the ACM SIGMOD Conference on Management of Data, California, USA*, May 2001.

[9] D. Dinh Kha, M. Yoshikawa, S. Uemura. An XML Indexing Structure with Relative Region Coordinate. *Proceedings of the International Conference on Data Engineering, Heidelberg, Germany*, pages 313–320, April 2001.

[10] D. Dinh Kha, M. Yoshikawa, S. Uemura. A Structural Numbering Scheme for XML Data. *EDBT workshop on XML Data Management (XMLDM), Prague.*

Lecture Notes of Computer Science 2490: XML-Based data Management and Multimedia Engineering, pages 91–108, 2002.

[11] D. Dinh Kha, M. Yoshikawa, S. Uemura. Application of rUID in Processing XML Queries on Structure and Keyword. *Proceedings of International Conference on Database and Expert Application, Aix-en-provence, France. Lecture Notes of Computer Science 2453*, pages 279–289, 2002.

[12] D. Florescu, D. Kossmann. A Perfomance Evaluation of Alternative Mapping Schemes for Storing XML data in Relational Database. *Technical Report 3680 INRIA http://rodin.inria.fr/dataFiles/FK99.ps*, May 1999.

[13] D. Florescu, I. Manolescu, D. Kossmann. Integrating Keyword Search into XML Query Processing. *Proceedings of the International WWW Conference, Elsevier, Amsterdam*, pages 119–135, May 2000.

[14] D. Shin. XML Indexing and Retrieval with a Hybrid Storage Model. *Knowledge and Information Systems*, 3:252–261, 2001.

[15] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, USA, 1997.

[16] F. Neven. Automata Theory for XML Researchers. *SIGMOD Record, 31(3), 2002.*, 2002.

[17] H. Garcia-Molina, J. D. Ullman, J. Widom. *Database System Implementation*. Prentice Hall, New Jersey, USA, 2000.

[18] H. Jang, Y. Kim, D. Shin. An Effective Mechanism for Index Update in Structured Documents. *Proceedings of CIKM, Kansas, USA*, pages 383–390, 1999.

[19] H. Jiang, H. Lu, W. Wang, B. C. Ooi. XR-Tree: Indexing XML data for Efficient Structural Joins. *Proceedings of the International Conference on Data Engineering, India*, 2003.

[20] H. Kaplan, T. Milo, and R. Shabo. A Comparison of Labeling Schemes for Ancestor Queries. *Proceedings of ACM-SIAM Symposium on Discrete Algorithms(SODA), USA*, pages 954–963, 2002.

[21] I. Tatarinov et al. Storing and Querying Ordered XML Using a Relational Database System. *Proceedings of the ACM SIGMOD, USA*, June 2002.

[22] I. Tatarinov, Zachary G. Ives, Alon Y. Halevy, Daniel S. Weld. Updating XML. *Proceedings of the ACM SIGMOD Conference on Management of Data, California, USA*, pages 413–424, May 2001.

[23] IBM. DB2 XML Extender. *http://www-3.ibm.com/software/data/db2/extenders/xmlext/*, 2002.

[24] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proceedings of the International Conference on Very Large Databases, Edinburgh, Scotland*, pages 302–314, 1999.

[25] M. J. Carey, D. J. DeWitt, J. E. Richardson, E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. *Proceedings of the International Conference on Very Large Databases, Kyoto, Japan*, pages 91–100, 1986.

[26] M. Yoshikawa, T. Amagasa, T. Shimura, S. Uemura. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transation on Internet Technologies*, 1(1):110–141, 2001.

[27] N. Bruno, N. Koudas, D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. *Proceedings of SIGMOD, USA*, pages 310–321*, 2002.

[28] Oracle company. Oracle 9*i*. *http://otn.oracle.com/products/oracle9i/content.html*, 2002.

[29] P. Buneman, S. Davidson, M. Fernandez, D. Suciu. Adding Structure to Unstructured Data. *Proceedings of the International Conference on Database Theory, Greece*, pages 336–350, 1997.

[30] P. F. Dietz. Maintaining order in a link list. *Proceedings of the Fourteenth ACM Symposium on Theory of Computing, California*, pages 122–127, May 1982.

[31] Q. Li, B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *Proceedings of the International Conference on Very Large Databases, Roma, Italy*, pages 361–370, September 2001.

[32] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. *Proceedings of the International Conference on Very Large Databases*, pages 436–445, September 1997.

[33] R. Sacks-Davis, T. Dao, J. A. Thom, J. Zobel. Indexing Documents for Queries on Structure, Content and Attributes. *Proceedings of International Symposium on Digital Media Information Base (DMIB), Nara, Japan,* pages 236–245, 1997.

[34] S. Abiteboul, H. Kaplan, and T. Milo. Compact Labeling Schemes for Ancestor Queries. *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA), USA*, pages 547–556, 2001.

[35] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. *Proceedings of ICDE, USA*, pages 141–148, 2002.

[36] S. Chien et al. Efficient Structural Joins on Indexed XML Documents. *Proceedings of VLDB, Hongkong*, 2002.

[37] S. Chien, V. J. Tsotras, C. Zaniolo, D. Zhang. Storing and Querying Multiversion XML Documents using Durable Node Numbers. *Proceedings of the International Conference on Web Information Systems Engineering, Kyoto, Japan*, pages 270–279, December 2001.

[38] T. Milo, D. Suciu. Index Structures for Path Expression. *Proceedings of the International Conference on Database Theory*, pages 277–295, 1999.

[39] Tobin J. Lehman, Bruce G. Lindsay. The Starburst Long Field Manager. *Proceedings of the International Conference on Very Large Databases, Amsterdam, Netherlands*, pages 375–383, 1989.

[40] Torsten Grust. Accelerating XPath Location Steps. *Proceedings of the ACM SIGMOD Conference on Management of Data, Wisconsin, USA*, pages 109–120, June 2002.

[41] University of California, Berkeley. Berkeley Database. *http://www.sleepycat.com/*, 2002.

[42] Wang Wei et al. PBiTree Coding and Efficient Processing of Containment Join. *Proceedings of the ICDE, India*, 2003.

[43] World Wide Web Consortium. Overview of SGML Resources. *http://www.w3.org/MarkUp/SGML/*.

[44] World Wide Web Consortium. HyperText Markup Language (HTML) Home Page. *http://www.w3.org/MarkUp/*.

[45] World Wide Web Consortium. Scalable Vector Graphics (SVG) 1.1 Specification. *http://www.w3.org/TR/SVG11/*.

[46] World Wide Web Consortium. Document Object Model (DOM) Level 1 Specification. *http://www.w3.org/TR/REC-xml*, 1998.

[47] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/REC-xml*, 2000.

[48] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. *http://www.w3.org/TR/xpath*, 2000.

[49] World Wide Web Consortium. Extensible Stylesheet Language (XSL) Version 1.0. *http://www.w3.org/TR/xsl/*, 2001.

[50] World Wide Web Consortium. XML Linking Language (XLink) Version 1.0. *http://www.w3.org/TR/xlink/*, 2001.

[51] World Wide Web Consortium. XQuery 1.0: An XML Query Language. *http://www.w3.org/TR/xquery*, 2002.

[52] Y. K. Lee, S-J. Yoo, K. Yoon, P. B. Berra. Index Structures for structured documents. *ACM First International Conference on Digital Libraries, Maryland*, pages 91–99, March 1996.

List of Publications

Journal Papers

1. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, "*A Structural Numbering Scheme for Processing Queries by Structure and Keyword on XML Data*", IEICE Special Section on Information Processing Technology for Web Utilization, pp.361-372, Vol.E87-D, No.2, February 2004.

2. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, "*XML Content Update using Relative Region Coordinate*", IEICE Transactions on Information and Systems, Vol.E87-D, No.3, March 2004.

International Conference Papers

1. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, "*Application of rUID in Processing XML Queries on Structure and Keyword*", Proceedings of 13th International Conference on Database and Expert Systems Applications (DEXA), Lecture Notes in Computer Science 2453, pp.279-289, September 2002.

2. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, "*A Structural Numbering Scheme for XML Data Proceedings of the Workshop on XML-based Data Management*", XML Data Management Workshop (in conjunction with the Conference on Extending Database Technology), Lecture Notes in Computer Science 2490, pp.91-108, March 2002.

3. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, "*An XML Indexing structure with Relative Region Coordinate*", Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE), pp.313-320, April,2001.

Other Publications

1. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, "*Virtual Joins for XML Data*", NAIST Technical Report IS-TR2003012, November 2003.

2. Dao Dinh Kha, "*Update-robustness for XML Data*", Ph.D. Doctoral Poster, VLDB, August 2002 (accepted, not presented).

3. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, “*Processing XML Queries on Structure and Keywords using rUID in SKEYRUS*”, Database Workshop, July 2002.

4. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, “*Processing XML Queries on Structure and Keywords in SKEYRUS*”, NAIST Technical Report IS-TR2002010, June 2002.

5. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, “*An Effective Storage of XML data with Relative Region Coordinate*”, The 62nd National Convention IPSJ, March 2001.

6. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, “*Application of Relative Region Coordinate for XML Storage*”, Data Engineering workshop, March 2001.

7. Masatoshi Yoshikawa, Toshiyuki Amagasa, Dao Dinh Kha, Kenji Hatano, Hiroko Kinutani, Noboru Matoba, Junko Tanoue, Masahiro Watanabe and Shunsuke Uemura, “*On Two Query for Interfaces Genome XML Databases*”, Proceedings of IEEE Workshop on XML-Enables Wide Area Search in Bioinformatics (XEWA), December 2000.

8. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura, “*An XML Indexing structure with Relative Region Coordinate*”, NAIST Technical Report IS-TR2000012, December 2000.

Appendix

A B$^+$-tree: an example

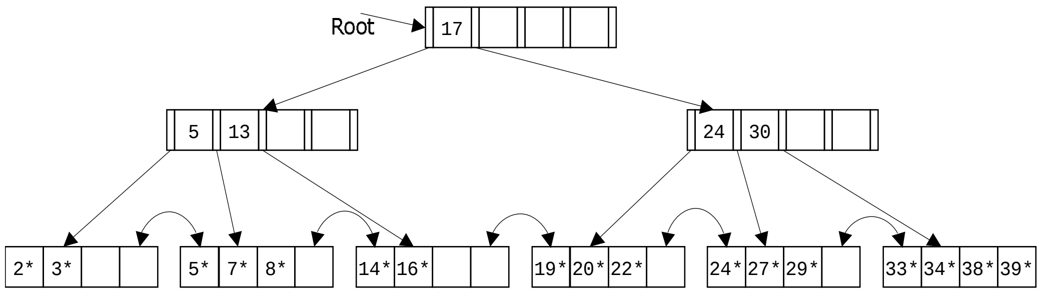


Figure 6.1. Example of a B$^+$ Tree, Order is equal 2.

B-trees, introduced by Bayer (1972) and McCreight, m-ary balanced trees the structure of which allows records to be inserted, deleted, and retrieved with guaranteed worst-case performance. An n-node B-tree has height $\mathcal{O}(\lg n)$, where lg is the logarithm to base 2. A B-tree organizes its blocks into a tree. The root is either a tree leaf or has at least two children. Each node (except the root and tree leaves) has at least half capacity, i.e. between $\lceil m/2 \rceil$ and m children, where $\lceil x \rceil$ is the ceiling function. In a B-tree, each path from the root to a tree leaf has the same length. In B$^+$ tree, leaves are connected by pointers to provide sequential access. An example of B$^+$-tree is depicted in Figure 6.1.

B R-tree: an example

R-Tree is a spatial access method, i.e. a data structure to search for lines, polygons, etc, which splits space with hierarchically nested, and possibly overlapping boxes. Objects are indexed in each box which intersects them. The tree is height-balanced. In a R-Tree, the root node has at least two children nodes unless it is a leaf. Every node contains between m and M children unless it is the root node. All leaves appear on the same level since the tree is balanced. Entries in non-leaf node contain a child-pointer to the address of a lower node in the R-tree and the bounding rectangle of this entry.

Entries in a leaf node contain a tuple-identifier referring to the tuple in the database and the bounding rectangle of this entry. An example of R-tree and the data indexed by it are depicted in Figure 6.2.

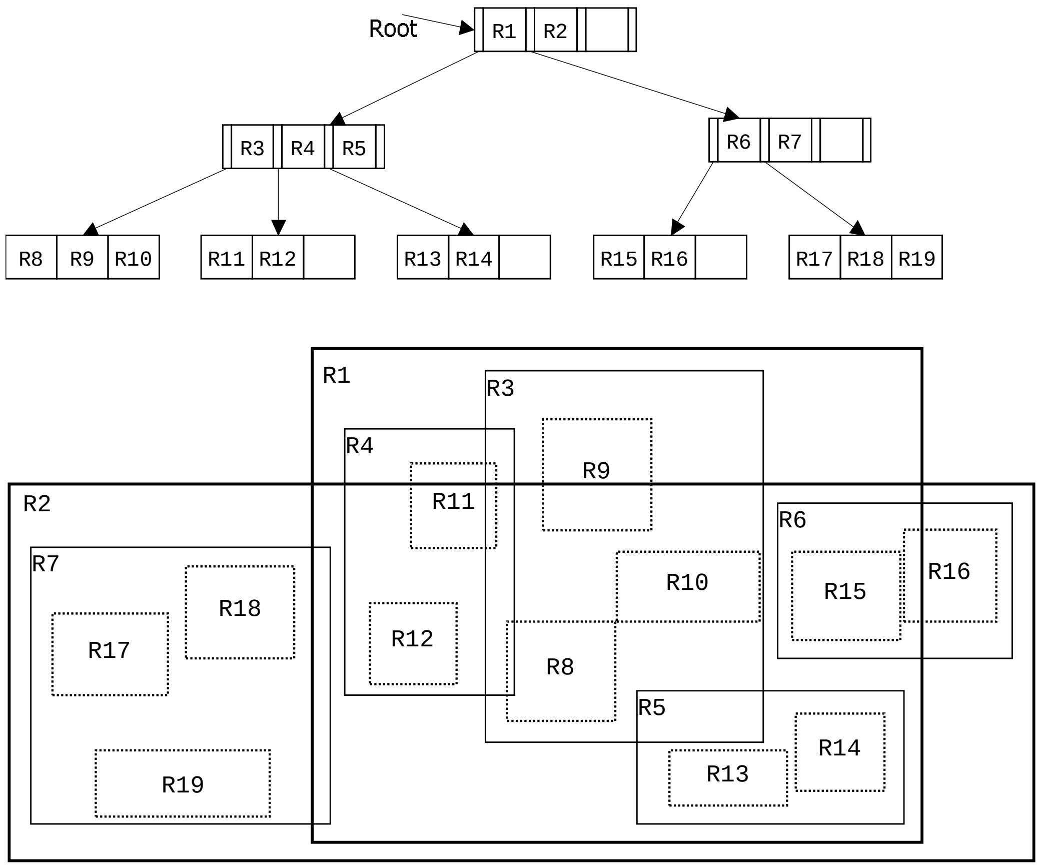


Figure 6.2. Example of a R-Tree and its spacial objects.

C The DTD of the datasets *I*

```
<!ELEMENT personnel (person)+>
<!ELEMENT person (name,email*,note?,link?,person*)>
<!ATTLIST person id ID #REQUIRED note CDATA #IMPLIED
      contr (true | false) "false" salary CDATA #IMPLIED>
<!ELEMENT name (family, given)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT note (#PCDATA)>
<!ELEMENT link EMPTY>
<!ATTLIST link manager IDREF #IMPLIED subordinates IDREFS #IMPLIED>
```

D The DTD of XMark datasets

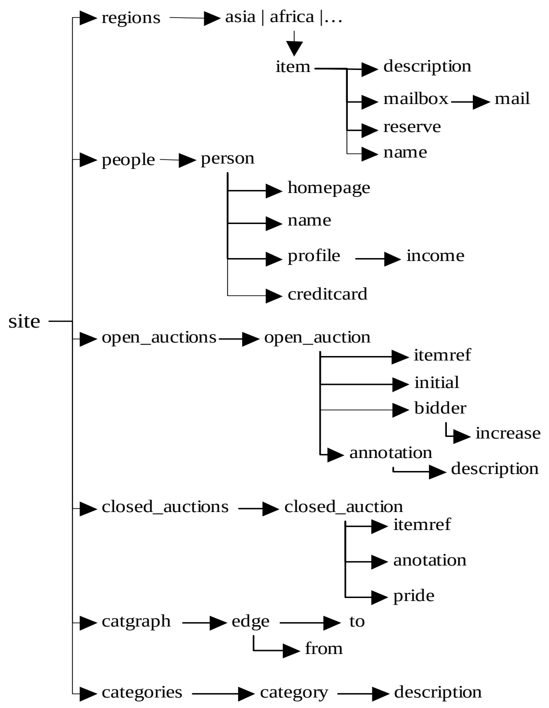


Figure 6.3. The sketch of the `auction.dtd` for XMark dataset.